Information and Software Technology 115 (2019) 58–78

Contents lists available at ScienceDirect

Information and Software Technology

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

Bug report severity level prediction in open source software: A survey and research opportunities

Luiz Alberto Ferreira Gomes[a,*], Ricardo da Silva Torres[b], Mario Lúcio Côrtes[b]

[a] Institute of Exact Sciences and Informatics (ICEI), Pontifical Catholic University of Minas Gerais, Poços de Caldas, Brazil
[b] Institute of Computing (IC), University of Campinas (UNICAMP), Campinas, Brazil

ARTICLE INFO

Keywords:
Software maintenance
Bug tracking systems
Bug reports
Severity level prediction
Software repositories
Systematic mapping
Machine learning

ABSTRACT

Context: The severity level attribute of a bug report is considered one of the most critical variables for planning evolution and maintenance in Free/Libre Open Source Software. This variable measures the impact the bug has on the successful execution of the software system and how soon a bug needs to be addressed by the development team. Both business and academic community have made an extensive investigation towards the proposal methods to automate the bug report severity prediction.
Objective: This paper aims to provide a comprehensive mapping study review of recent research efforts on automatically bug report severity prediction. To the best of our knowledge, this is the first review to categorize quantitatively more than ten aspects of the experiments reported in several papers on bug report severity prediction.
Method: The mapping study review was performed by searching four electronic databases. Studies published until December 2017 were considered. The initial resulting comprised of 54 papers. From this set, a total of 18 papers were selected. After performing snowballing, more nine papers were selected.
Results: From the mapping study, we identified 27 studies addressing bug report severity prediction on Free/Libre Open Source Software. The gathered data confirm the relevance of this topic, reflects the scientific maturity of the research area, as well as, identify gaps, which can motivate new research initiatives.
Conclusion: The message drawn from this review is that unstructured text features along with traditional machine learning algorithms and text mining methods have been playing a central role in the most proposed methods in literature to predict bug severity level. This scenario suggests that there is room for improving prediction results using state-of-the-art machine learning and text mining algorithms and techniques.

1. 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 information of severity level is recognized as a critical variable for prioritizing and planning how bug reports will be dealt with [3]. It measures the impact the bug has on the successful execution of the software system and defines how soon the bug needs to be addressed [4]. However, the severity level assignment remains mostly a manual process, which relies only on the experience and expertise of the person who has opened the bug report [1,3,4].

Moreover, the number of bug reports in large and medium software FLOSS projects is frequently very large [5]. For example, Eclipse project had 84,245 bug reports opened from 2013 to 2015 alone, whereas Android project had over 107,456, and JBoss project had over 81,920.

* Corresponding author.
E-mail addresses: luizgomes@pucpcaldas.br (L.A.F. Gomes), rtorres@ic.unicamp.br (R.d.S. Torres), cortes@ic.unicamp.br (M.L. Côrtes).

https://doi.org/10.1016/j.infsof.2019.07.009
Received 11 September 2018; Received in revised form 26 June 2019; Accepted 30 July 2019
Available online 31 July 2019

Therefore, a manual assignment of severity level may be a quite subjective, cumbersome and error-prone process, and a wrong decision throughout 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 the evident importance of bug report severity information for the planning of FLOSS maintenance, both business and academic community have demonstrated great interest, and plenty of research has been done in this field. However, there are few mapping study reviews well characterizing this area, positioning existing works and measuring current progress. Although these existing works helped to understand the challenges and opportunities in this research area [1,6], they suffer from one shortcoming: existing reviews lack in-depth coverage of the different dimensions of bug report severity prediction. Our current mapping study review fills this gap, by investigating and analyzing relevant papers — published from 2010 to 2017 — related to bug report severity prediction. To the best of our knowledge, this is the first review to provide a detailed document with the analysis of more than ten relevant aspects of the surveyed experiments, including: granularity of output class, FLOSS repositories, features and feature selection methods, text mining methods, machine learning algorithms, performance evaluation measures, sampling techniques, statistical tests, experimental tools, and type of solution.

A mapping study review aims to characterize the state-of-the-art on bug report severity prediction. This sort of study provides a broad overview of a research area to determine whether there is research evidence on a particular topic. According to Kitchenham et al. [7] and Petersen et al. [8], results yielded by a mapping study review help recognizing gaps to suggest future research and provide a direction to engage in new research activities appropriately. The key contributions of our review are three-fold:

- It proposes a categorization scheme to organize the experiments reported in papers on bug report severity prediction. For example, it provides a taxonomy to categorize the severity prediction problem type based on the granularity severity level which one wishes to predict.
- It provides an overview of the state-of-the-art research by summarizing patterns, procedures, and methods employed by researches to predict bug report severity. For example, it indicates which are the most used machine learning algorithms in reported experiments.
- It discusses challenges, and open issues and future directions in this research field to share the vision and expand the horizon of bug report severity prediction.

This paper is organized as follows: Section 2 provides the basic information background necessary to understand the research area. Section 3 presents related work. Section 4 describes our research method. Section 5 presents our results. Section 6 presents final findings and discussion. Finally, Section 8 concludes the paper.

2. Terminology and background

This section presents an overview on general concepts necessary to understand this research area, namely Bug Tracking Systems, Machine Learning (algorithms, feature selection methods, evaluation measures, and sampling methods), Text Mining, and Statistical Tests. More specific concepts will be detailed as they are cited in the document.

2.1. Bug tracking systems

Bug Tracking Systems (BTS) [4] 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.

2.1.1. Bug report

Although there is no agreement on the terminology or the amount of information that users must provide to fill a typical bug report (illustrated in Fig. 1, the example refers to a bug report extracted from Bugzilla of Eclipse project), they often describe their needs in popular BTS (e.g., Bugzilla, Jira, and Redmine) [3] providing information about at least attributes shown in Table 1.

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. Fig. 2a shows an example of such a state machine, with a typical set of states a bug report can hold during its lifecycle in a BTS. Initially, a bug report is in *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 Fig. 2a, state transitions may occur throughout the bug report lifecycle. All changes occurred in a bug report are stored in a repository, keeping valuable historical information about a particular software.

Both users and development team members can define or redefine the severity level for a bug during the lifecycle. Fig. 2b illustrates Bugzilla guidelines for assigning bug severity level. The Figure shows that such choices should be based on an affirmative answer to a question which characterizes a severity level appropriately. Also, the Figure indicates that *Trivial* and *Blocker* are lower and higher, respectively.

To predict severity level, researchers sometimes aggregate these levels in severe (blocker, critical, and major) and non-severe (normal, minor and trivial) to work with a coarse-grained classification problem. Furthermore, some of them ignore the default severity level (often “normal”) because they consider this level as a choice made by users when they are not sure about the correct severity level. Other studies choose to predict a bug report severity level as *blocking* or *non-blocking* bug. A blocking bug is one that prevents other bugs to be fixed [10].

2.1.2. Bug severity

The terminology of bug severity can be expressed by various terms depending on of the BTS used. Meaning of bug severity from three popular BTS are described next:

- **Bug Severity in Bugzilla** indicates how severe the problem is – from blocker (“application unusable”) to trivial (“minor cosmetic issue”). Also, this field can be used to indicate whether a bug is an enhancement request.[1] On the other hand, the priority defines how urgent a bug needs to be fixed. In Bugzilla, the combination between priority and severity defines the importance of a Bug.
- **Bug Severity in Jira** is referred to as “priority” and indicates how important the bug - from blocker (“highest priority”) to minor (“low priority”) is in relation to others bugs.[2]
- **Bug Severity in Google Issue Tracker** is also referred to as “priority” and indicates how priority the bug is - from P0 (“needs to be addressed immediately”) to P4 (“needs to be addressed immediately eventually”) in relation to others bugs.[3]

[1] The Bugzilla Guide on https://www.bugzilla.org/docs/4.2/en/html/bug_page.html (As of september 2018).

[2] Administering Jira Sever Documentation on https://confluence.atlassian.com/adminjiraserver/defining-priority-field-values-938847101.html (As of september 2018).

[3] Google Issue Tracker Documentation on https://developers.google.com/issue-tracker/concepts/issues (As of september 2018).

Fig. 1. A typical bug report (https://issues.apache.org/jira/browse/HADOOP-8855, as of september 2018).

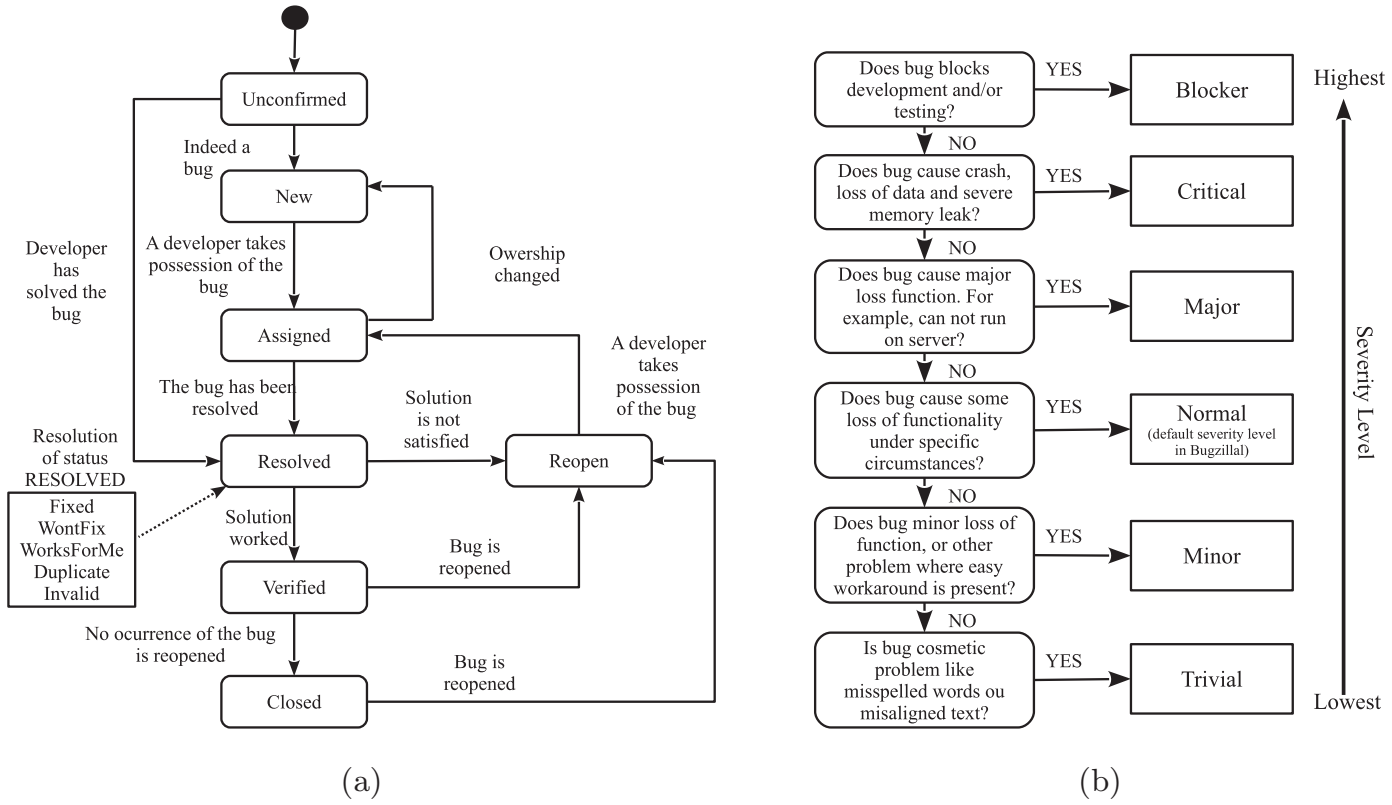

Fig. 2. (a) The bug report lifecycle according to Zhang et al. [9]. (b) Bugzilla guideline for bug report severity level assignment (https://www.bugzilla.org, as of september 2018).

Table 1
Commom attributes in a bug report form.

Type	Type of report (e.g., bug, improvement, and new feature)
Summary	Short description of 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.
Severity	Level of severity of report (e.g., blocker, critical, major, minor and trivial).

Inexperienced users my consider bug severity as an information that is often difficult to understand or it is often mistakenly perceived as a particular priority. Average users have limited technical experience, which hampers them to correctly assess the severity of a bug when they have to report bug [11]. On the other hand, to developers, bug severity measures the impact or how severe a bug is for their software. They can, therefore, use the severity with other available information to prioritize bug fixing tasks [12].

2.2. Machine learning

Machine Learning(ML) [13] is an application of Artificial Intelligence (AI) [76] 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). A predictive algorithm builds a model based on historical training data and uses this model to predict, from the values of input at-

tributes, 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.

On the other hand, a descriptive algorithm explores or describes a dataset. There is not an output label associated with a sample. Data clustering and pattern discovery are two examples of descriptive tasks. Bug report severity prediction is considered a classification problem; therefore, more detailing of descriptive algorithms are outside the scope of this paper.

2.2.1. ML algorithms

An ML algorithm works over a dataset, which contains many samples or instances x_i, where $i = \{1 \dots n\}$. Each instance is composed of $\{x_{i1}, x_{i2}, \dots, x_{id}\}$ input attributes or independent variables, where $d = \{1 \dots m\}$, 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. The most traditional ML classification algorithms are k-Nearest Neighbors, Näive Bayes, Decision Tree, Neural Networks, Random Forest, and Support Vector Machine. In practice, they can be applied for both classification and regression tasks. However, this mapping study review regards them only in the classification scenario. Next, a brief description of each one is presented [14]:

- **k-Nearest Neighbors** (KNN) process the available instances (or neighbors) in a dataset and classifies a new instance based on its similarity measure to the k-nearest neighbors. Usually, k-NN algorithms utilize a Euclidean distance to quantify the proximity of neighbors. To calculate this distance, each feature vector of each instance in a dataset should represent a point of an n-dimensional space [77].
- **Näive 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 $P(C_k|x)$. Näive Bayes classifiers assume that given the class variable, the value of a particular feature is independent of the value of any other feature [75].
  Multinomial Näive Bayes (MNB) [12,78] differently of Näive Bayes, MNB decides the output class, not only by the presence or absence of terms, but also by the number of occurrences of the terms in a document which is represented as a multinomial distribution of words. MNB treats this document as an ordered sequence of word occurrences, with each word occurrence as an independent trial. Formally, each $P(C_k|x)$ is a multinomial distribution, rather than some other distribution [15].
- **Decision Tree** consists of a collection of internal nodes and leaf nodes in a tree, organized in a hierarchical model. Each internal node represents a feature, and each leaf node corresponds to a class label. The decision tree classifiers organize a series of test questions and conditions in a tree structure. A decision tree represents the model capable of guiding the decision making on the determination of the class to which an individual belongs.
- **Neural Network** is a learning algorithm that is inspired by the structure and functional aspects of biological neural networks [16]. It 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 [17].
- **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. 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 [18,74].
- **Random Forest** [19] 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 is based on the ruling of the majority of the forming trees.

Table 2
Example of confusion matrix.

		Predicted	
		Negative	Positive
Actual	Negative	TN	FP
	Positive	FN	TP

2.2.2. Feature selection methods

Feature selection is the process of choosing a subset of features that better contribute to the accuracy of a predictive model. Three typical feature selection methods are described next [20,80]:

- **Information Gain (IG)**: this method measures the number of bits of information obtained for category prediction by knowing the presence or absence of a feature in a dataset.
- **Chi-square (CHI)**: this method measures the lack of independence between a feature f and category c_i and can be compared to the chi-square distribution with one degree of freedom.
- **Correlation Coefficient (CC)**: this method defines the correlation coefficient of feature f with a category c_i.

2.2.3. Evaluation measures

Accuracy, precision, recall, and F-measure are four measures commonly used to evaluate the performance of prediction models [21]. The computation of the values of these measures are based on a *confusion matrix*, as showed in Table 2. TP, TN, FP, and FN stand for the number of true positive, true negative, false positive, and false negative classified samples, respectively. More formally, let the matrix showed in Table 2 be a confusion matrix, which encodes the number of instances according to the following definition [22]:

- **Positive (P)**: The instance is positive (e.g., a bug is severe)
- **Negative (N)**: The instance is not positive (e.g., a bug is not severe)
- **True Positive (TP)**: an instance that is positive and was correctly predicted by classifier as positive.
- **True Negative (TN)**: an instance that is negative" and was correctly predicted by the classifier as negative.
- **False Positive (FP)**: an instance that is negative and was incorrectly predicted by the classifier as positive.
- **False Negative (FN)**: an instance that is positive and was incorrectly predicted by the classifier as negative.

Common Measures derived from confusion matrix are described next [23]:

- **Accuracy** is the percentage of correctly classified observations among all observations:

$$Accuracy = \frac{TP + TN}{P + N} \tag{1}$$

  where P is the total of positive class instances, N is the total of negative class instances, TP is the number of true positives, and TN is the number of true negatives.
- **Precision** is the percentage of correctly classified observations among all observations that were assigned to the class by the classifier. It can be viewed as a measure of classifier exactness. A low precision can also indicate a large number of false positives. More formally recall is defined as:

$$Precision = \frac{TP}{TP + FP} \tag{2}$$

  where TP is the number of true positives, and FP is the number of false positives.

- **Recall** of a classification method can be defined as the percentage of correctly classified observations among all observations belonging to that class. It can be viewed as a measure of a classifiers completeness. A low recall indicates many false negatives in testing classification step. More formally recall is defined as:

$$Recall = \frac{TP}{TP + FN} \tag{3}$$

where TP is the number of true positives, and FN is the number of false negatives.

- **F-measure** is a harmonic mean of the precision and recall [21]. F-measure can be calculated as:

$$F\text{-}measure = \frac{2 \times (precision \times recall)}{(precision + recall)} \tag{4}$$

Receiving Operating Characteristics (ROC) is an alternative measure to evaluate binary classifiers. A ROC curve [23] is a bidimensional chart, where the X-axis represents false positives, and the Y-axis represents true positives. The Area Under ROC Curve (AUC), ranging between 0 and 1, is used to assess the performance of ML algorithms. An algorithm outperforms another one if its AUC value is closer to 1.

Performance issues in ML algorithms, which degrade bug report prediction, include among others [24,25]: (i) data imbalance and (ii) high dimensionality. In the real world, datasets extracted from bug tracking systems are naturally imbalanced. Conventional machine learning algorithms do not accurately perform well when they handle imbalanced datasets. Some of them, for example, do not take into account the class distribution/proportion or balanced class, producing results biased towards classes which have the largest number of samples [24]. The vector extraction approach most used for bug severity prediction is Bag-of-Words (BoW). BoW is often used to extract features from three unstructured text fields of a bug report form: summary, description, and comment. Despite its popularity and simplicity, BoW may lead to a model with high dimensions of features, which lead to the need of handling the severe effectiveness and efficiency issues related to the "curse of dimensionality." Standard ML algorithms usually struggle to define hyperplanes to separate correctly classes in high dimensional spaces [25]. Also, storage requirements tend to be more costly when handling samples with high dimensionality. Another issue refers to the lack of labeled datasets to train machine learning methods. The assignment of labels is usually a time-consuming and error-prone task. The lack of labeled training samples affect the capacity of generalization of algorithms [26].

The fine tuning of hyper-parameters values is very important in the machine learning optimization area, as each algorithm has its own set of parameters and the adjustment of these values can greatly impact in 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% [27].

The tuning of hyper-parameters is considered an iterative activity that occurs during the training phase of an ML building process. In a typical protocol, initially, researchers manually pick one or more ML algorithms. After that, they set the hyper-parameters candidate values for each algorithm (e.g., the value of k for KNN algorithm). Three common procedures are used by researchers for setting hyper-parameters [28]: (i) default values specified in the software packages implementation, (ii) manual configuration based on recommendations from 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 changing the hyper-parameters candidate values and/or choosing other ML algorithm, researchers should train the model again until the predicting model achieves satisfactory prediction accuracy. When this goal is reached, the machine learning model building process can be considered complete.

2.2.4. Sampling methods

The evaluation of the supervised method effectiveness is mainly based on two datasets with labeled samples, one for training the predictive model and the other for testing this model. Assessing performance of a predictive model using the same dataset for training and testing is not recommend and may yield misleading optimistic estimations [23]. In order to obtain more reliable predictive estimates, resampling methods can be used to split the entire dataset into a training dataset and a testing dataset. Such methods according to Marsland et al. [14] include:

- **Holdout** splits a dataset into a ratio of p for training and $(1 - p)$ for testing.
- **Cross-Validation (CV)** divides a dataset into k folds. In each iteration, it saves a different fold for testing and uses all the others for training.
- **Bootstrap** generates r training subsets from the original dataset. It randomly samples instances with replacement from this set. Unselected cases make up the test subset. The result is the average performance observed for each test subset.

2.2.5. Statistical tests

Many scenarios require running several ML algorithms to choose the best predictive model. Even though the performance of these algorithms may be shown to be different on specific datasets, it needs to be confirmed whether the observed differences are statistically significant and not merely coincidental [23]. In this situation, conducting statistical tests is a recommended practice for reliable comparison between predictive models under investigation [29,73]. A brief description of four common statistical tests is provided:

- **T-Test** [23] is a parametric statistical hypothesis test that can be used to assess whether the means of two groups are statistically different from each other.
- **Wilcoxon signed-rank test** [30] is a non-parametric statistical hypothesis test that can be used to determine whether two dependent samples, selected from the population, have the same distribution.
- **Proportion test** [31] is a parametric statistical hypothesis test that can be used to assess whether or not a sample from a population represents the true proportion of the entire population.
- **Shapiro Wilk test** [31] is a parametric statistical hypothesis test that can be used to test whether a sample $x_1, \ldots, x_n$ came from a normal distribution of the population.

2.3. Text mining

The common ML algorithms cannot directly process unstructured text (e.g., *summary* and *description* fields from 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 [21]. It is composed of three primary activities [22]:

- **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").

2.3.1. Vector extraction models

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) [21]. 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).

2.3.2. Vector weighting models

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.

2.3.3. Document similarity methods

Measuring the similarity between words, sentences, paragraphs, and documents is an important component in text mining tasks [32]. Next, a brief description of common document similarity methods:

- **Topic Model** [33] is a statistical model to identify "topics" from a collection of documents. Each topic includes the topic terms which appear in the documents and each document may belong to one or more topics.
- **BM25** [34] is a ranking function used by search engines to rank matching documents according to their relevance to a given search query.
- **BM25F** [3] is a function to evaluate the similarity between two structured documents. Each of the fields in these documents can be assigned a different weight to denote its importance in measuring the similarity.
- **BM25ext** [33] is an extension of similarity function BM25. While BM25 computes the similarity of a short document with a long document, BM25ext computes the similarity between long documents.
- **Kullback-Leibler (KL)** [35] divergence measures the difference between two probability over the same variable.
- **Cosine similarity** [32] is a measure of similarity between two vectors defined in terms of the cosine of the angle between them.
- **TS-SS** [36] is a method to measure the similarity level among documents based on the geometrical similarity of keyword(s).

3. Related work

To the extent of our knowledge, only two papers [1,6] have reviewed the literature related to bug report severity prediction. Cavalcanti et al. [1] performed a review of 142 papers, published between 2000 to 2012, that investigated challenges and opportunities for software change repositories. Just seven of them are related to change request prioritization, which is defined in Bugzilla by two fields: *priority* and *severity*. Four out of them addressed bug report severity prediction. Only two papers (Lamkanfi et al. [4] and Lamkanfi et al. [12]), however, addressed severity prediction on FLOSS projects. That mapping review shows that all of seven papers used some Information Retrieval Model: one paper used the vector space representation (binary and term count), and all seven used TF-IDF. It also shows that all papers implement learning techniques, such as SVM (4 out of 7), Decision tree (2 out of 7), k-NN (2 out of 7), Naïve Bayes (3 out of 7), and Multinomial Naïve Bayes (1 out of 7).

The review presented by Uddin et al. [6], in turn, surveyed published papers in bug prioritization. The authors reviewed and analyzed in depth 32 distinct papers published between 2003 and 2015. The aim of that analysis was "to summarize the existing work on bug prioritization and some problems in working with bug prioritization". That work categorized research initiatives according to ML algorithms, evaluation measures, data sets, researchers, and publication venue. It can be worth to note that the authors investigated only eight papers about predicting of severity level.

Although these reviews present relevant results for researchers in this area, we can observe three major shortcomings: First, none of the existing reviews specifically focused on "bug severity prediction". Both [1] and [6] focused on "bug **priority** prediction". Although the severity may be strongly related to prioritization activities, in Bugzilla, the most used BTS in severity prediction experiments, priority and severity are not interchangeable concepts. In our mapping review, we have covered the state of the art in bug severity prediction, investigating deeply 27 papers, published from 2010 to 2017.

Second, existing reviews characterized "bug severity prediction" considering much less dimensions of analysis. In our review, 12 dimensions are considered, including among others, employed machine learning approaches, text mining methods, sampling techniques, statistical tests, and software tools. The work of Cavalcanti et al. [1], for example, focused only on two dimensions: vector space representation models and machine learning approaches. Uddin et al. [6], in turn, were most interested in the employed machine learning approaches, evaluation metrics, and data sets dimensions. In summary, in our review, papers in the area are analyzed in an integrated manner for the first time, by taking into account a broader set of research dimensions.

Third, while each related review compared papers based only on few categories using few tables and charts, the current mapping review provides summarized information by considering 12 categories, analyzed by means of 14 tables and 11 upset charts. UpSet [37] is a novel visualization technique for the quantitative analysis of sets, their intersections, and aggregates of intersections. This chart focuses on creating task-driven aggregates, communicating the size and properties of aggregates and intersections, and a duality between the visualization of the elements in a dataset and their set membership. We believe that the use of these visual artifacts for comparing research initiatives in the area of bug severity prediction allows for the identification, more easily, of patterns and trends in the area.

In addition to the two literature reviews discussed before, Chaturvedi et al. [38] classified datasets, techniques, and tools in mining software engineering. They examined around 150 papers published from 2007 to 2013 in proceedings of the conferences on Mining Software Repositories (MSR) and other related conferences and journals. Furthermore, the authors categorized the selected tools based on newly developed, traditional data mining tools, prototype developed, and scripts. They showed that most of the researchers used already developed tools for data mining tasks in software engineering repositories. These tools were primarily used for data extraction from repositories, required data filtering, pattern finding, learning, and prediction. In their review, only one out of selected papers, Lamkanfi et al. [4] was related to bug severity prediction.

4. Research method

This section describes the research method used in this mapping review for identifying and analyzing relevant papers. It is mainly based on the software engineering systematic literature review guidelines and recommendations proposed by Kitchenham et al. [7,72] Brhel et al. [39], Souza et al. [40], and Petersen et al. [8]. These authors suggest a process with three main steps: planning, conducting, and reporting. In the planning step, they recommend defining a review protocol that includes a set of research questions, inclusion and exclusion criteria, sources of papers, search string, and mapping procedures. In the conducting step, they recommend selecting and retrieving papers for data extraction. Finally, in the reporting step, they recommend analyzing the results used to address research questions defined before.

4.1. Research questions

The main goal of this mapping study review is to provide a current status of the research on bug report severity prediction in FLOSS. To ensure an unbiased selection process, this review addresses the following research questions.

- **RQ_1. When and where have the studies been published?** This research question gives to the researcher a perception whether the topic of this mapping review seems to be broad and new, providing an overview about where and when papers were published.
- **RQ_2. What FLOSS are the most used as report sources in experiments for bug report severity prediction?** This research question names the FLOSS used as report source in problems of bug report severity prediction. This overview can be useful for researchers that intend to accomplish new initiatives in this area, and can also motivate the adoption of new FLOSS in future research to bridge existing gaps.
- **RQ_3. Was bug report severity prediction most addressed as either a fine-grained label or coarse-grained label prediction problem?** This research question investigates whether bug report severity prediction was handled as a fine-grained label (i.e., multi-label) or as a coarse-grained label (i.e., two-label or three-label) prediction problem. Understanding each solution provided for each prediction problem type is essential for guiding the selection of suitable machine learning algorithm.
- **RQ_4. What are the most common features used for bug report severity prediction?** This question investigates features used for bug report severity prediction in FLOSS. It can help to map which features have been considered more effective for this prediction problem.
- **RQ_5. What are the most common feature selection methods used for bug report severity prediction?** This question complements the previous one, looking for feature selection methods employed in the papers for bug report severity prediction in FLOSS. Also, this RQ allows for identifying which feature selection approaches have been considered more suitable for this prediction problem.
- **RQ_6. What are the most used text mining methods for bug report severity prediction?** This research question indicates the main text mining approaches (including vector extraction models, vector weighting models, and vector similarity/dissimilarity functions) used to extract features from textual fields of bug reports. It can guide researchers in the task of selecting text mining methods more suitable to be applied in bug report severity prediction or similar problems.
- **RQ_7. What are the most used machine learning algorithms for bug report severity prediction?** This research question aims to identify the main ML algorithms, which have been adopted for bug report severity prediction in FLOSS. Also, this RQ can be useful for any researcher that intends to accomplish further initiatives in this area, as well as to guide him/her in search of other algorithms to advance the state of the art.
- **RQ_8. What are the measures typically used to evaluate ML algorithms performance for bug report severity prediction?** This research question aims to find out which measures are used to evaluate ML algorithms performance. It can be helpful to identify the more appropriate measures to evaluate ML algorithms performance for bug report severity prediction.
- **RQ_9. Which sampling techniques are applied most frequently to generate more reliable predictive performance estimates in severity prediction of a bug report?** This question complements the previous one, investigating sampling techniques, which have been used to generate more reliable and accurate ML models. It can be useful to analyze sampling techniques, which might be more suitable to improve machine learning algorithms performance for bug report severity prediction.
- **RQ_{10}. Which statistical tests were used to compare the performance between two or more ML algorithms for bug report severity prediction?** The answer to this research question can be useful to identify which statistical tests were used in the context of bug report severity prediction. In addition, it allows grasping existing protocols for the comparison between two or more ML algorithms performance using statistical significance tests.
- **RQ_{11}. Which software tools were used to run experiments for bug report severity prediction?** This research question highlights the software tools most used to run experiments for bug reports severity prediction in FLOSS projects. It is useful to provide information for researchers and practitioners about technologies that could be used in their experiments.
- **RQ_{12}. Which solution types were proposed for the problem of bug report severity prediction?** This research question examines whether the most common solutions proposed in the selected studies are either online or off-line. This is important to separate the solutions that can be applied only in an experimental environment (off-line) from those that also can be deployed in a real scenario (online).

4.2. Paper selection

This section describes the selection process carried out through this mapping study review. It comprises four steps: (i) terms and searching string; (ii) sources for searching; (iii) inclusion and exclusion criteria; and (iv) data storage procedures.

4.2.1. Terms and search string

The base string was constructed from three main search terms related to three distinct knowledge areas: “open source project”, “bug report” and “severity predict.” To build the search string, terms were combined with “AND” connectors. The search string syntax was adapted according to particularities of each source (e.g., wildcards, connectors, apostrophes, and quotation marks) before it has been applied on three metadata papers: title, abstract, and keywords. Table 3 exhibits terms and final search string used in this mapping study review.

4.2.2. Sources

To accomplish this mapping study review, four electronic databases and one popular search engines were selected, as recommended by Kitchenham et al. [7]. Table 4 shows each selected search sources, as well as the search date, and the period covered by the search.

4.2.3. Inclusion and exclusion criteria

The inclusion criteria below allow the identification of papers in the existing literature.

- **IC_1.** The paper discusses bug report severity prediction in FLOSS projects.

On the contrary, the following exclusion criteria allow excluding all papers that satisfy any of them.

- **RQ_1.** The paper does not have an abstract;
- **RQ_2.** The paper is just published as an abstract;
- **RQ_3.** The paper is written in a language other than English;
- **RQ_4.** The paper is not a primary paper (e.g., keynotes);
- **RQ_5.** The paper is not accessible on the Web;

4.2.4. Data storage

The data extracted in the searching phase were stored into a spreadsheet that recorded all relevant details from selected papers. This spreadsheet supported classification and analysis procedures throughout this mapping study review.

Table 3
Search string.

Area	Search term
Bug report	"bug report"
Open source	"open source software"
Severity prediction	"severity predict"
Search string:	"open source software" AND "severity predict" AND "bug report"

Table 4
Search sources.

Source	Electronic address	Type	Search Date	Years covered
ACM	http://dl.acm.og	Digital library	Dec,17	2010–2017
IEEE	http://ieeexplore.ieee.org	Digital library	Dec,17	2010–2017
Google Scholar	http://www.scholar.google.com	Search engine	Dec,17	2010–2017
Science Direct	http://www.sciencedirect.com	Digital library	Dec,17	2010–2017
SpringLink	http://www.springerlink.com	Digital library	Dec,17	2010–2017

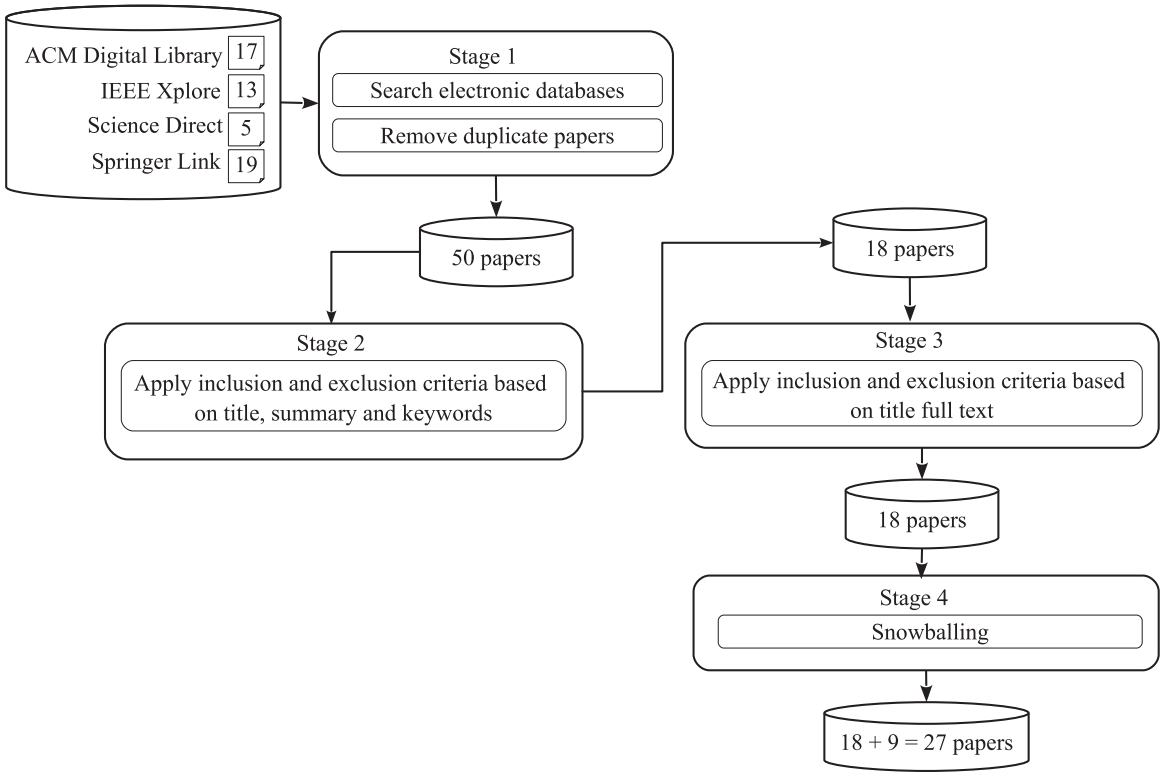

Fig. 3. Diagram of the four-stage paper selection process.

4.2.5. Assessment

According to Kitchenham et al. [7], the consistency of the protocol utilized in a systematic mapping should be reviewed to confirm that: "the search strings, constructed interactively, derived from the research questions"; "the data to be extracted will properly address the research question(s)"; "the data analysis procedure is appropriate to answer the research questions". In this research, the first author is a Ph.D. candidate running the experiments. The second and third authors conducted the review process.

4.3. Data extraction and synthesis

Fig. 3 illustrates the four-stage selection process carried out in the current mapping study review. In each stage, the sample size was reduced based on the inclusion and exclusion criteria. In the first stage, relevant papers were retrieved by querying the databases with the search string presented in Table 3. All database queries were issued in December 2017. This yielded a total of 54 initial papers: 13 from **IEEE Xplore**, 5 from **Science Direct**, 17 from **ACM Digital Library**, and 19 from **SpringerLink**. After removing four duplicates, we reduced (around 7%) the initial set of 50 papers. In the second stage, inclusion and exclusion criteria were applied over title, abstract, and keywords, reaching a set of 18 papers (reduction around 64%): 32 papers were rejected for not satisfying IC_1 (The paper discusses severity prediction of bug reports in FLOSS projects). In the third stage, exclusion criteria were applied considering the full text. However, no paper was rejected by taking into account these criteria.

In the fourth stage, the Snowballing (search for references) [7] activity was conducted, which resulted in 12 additional papers. After applying selection criteria over title, abstract, and keywords, 11 papers remained (reduction of 8.3% over the papers selected by snowballing). For these papers, selection criteria were applied considering the full text, and nine papers remained (reduction of approximately 18.2% over the 11 previously selected papers); EC_3 criterion eliminated one paper (the

Table 5
Selected papers.

ID	Bibliographic reference
[4]	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.
[12]	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.
[3]	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:10.1109/WCRE.2012.31.
[25]	C. Z. Yang, C. C. Hou, W. C. Kao, I. X. Chen, An empirical study on improving severity prediction of defect reports using feature selection, in: 2012 19th Asia-Pacific Software Engineering Conference, Vol. 1, 2012, pp. 240–249. doi:10.1109/APSEC.2012.144.
[41]	K. Chaturvedi, V. Singh, An empirical comparison of machine learning techniques in predicting the bug severity of open and closed source projects, International Journal of Open Source Software and Processes (IJOSSP) 4 (2) (2012) 32–59. doi:10.4018/jossp.2012040103.
[42]	C. Z. Yang, K. Y. Chen, W. C. Kao, C. C. Yang, Improving severity prediction on software bug reports using quality indicators, in: 2014 IEEE 5th International Conference on Software Engineering and Service Science, 2014, pp. 216–219. doi:10.1109/ICSESS.2014.6933548.
[35]	G. Yang, T. Zhang, B. Lee, Towards semi-automatic bug triage and severity prediction based on topic model and multi-feature of bug reports, in: 2014 IEEE 38th Annual Computer Software and Applications Conference, 2014, pp. 97–106. doi:10.1109/COMPSAC.2014.16.
[10]	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.
[43]	M. Sharma, M. Kumari, R. K. Singh, V. B. Singh, Multiattribute based machine learning models for severity prediction in cross project context, in: B. Murgante, S. Misra, A. M. A. C. Rocha, C. Torre, J. G. Rocha, M. I. Falc ao, D. Taniar, B. O. Apduhan, O. Gervasi (Eds.), Computational Science and Its Applications ICCSA 2014, Springer International Publishing, Cham, 2014, pp. 227–241. doi:10.1007/978-3-319-09156-3_17.
[44]	N. K. S. Roy, B. Rossi, Towards an improvement of bug severity classification, in: 2014 40th EUROMICRO Conference on Software Engineering and Advanced Applications, 2014, pp. 269–276. doi:10.1109/SEAA.2014.51.
[26]	R. K. Saha, J. Lawall, S. Khurshid, D. E. Perry, Are these bugs really “normal”?, in: 2015 IEEE/ACM 12th Working Conference on Mining Software Repositories, 2015, pp. 258–268. doi:10.1109/MSR.2015.31.
[9]	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.
[45]	G. Sharma, S. Sharma, S. Gujral, A novel way of assessing software bug severity using dictionary of critical terms, Procedia Computer Science 70 (2015) 632–639, proceedings of the 4th International Conference on Eco-friendly Computing and Communication Systems. doi:10.1016/j.procs.2015.10.059.
[46]	X. Xia, D. Lo, E. Shihab, X. Wang, X. Yang, Elblocker: Predicting blocking bugs with ensemble imbalance learning, Information and Software Technology 61 (2015) 93–106. doi:10.1016/j.infsof.2014.12.006.
[47]	S. Gujral, G. Sharma, S. Sharma, Diksha, Classifying bug severity using dictionary based approach, in: 2015 International Conference on Futuristic Trends on Computational Analysis and Knowledge Management (ABLAZE), 2015, pp. 599–602. doi:10.1109/ABLAZE.2015.7154933.
[48]	M. N. Pushpalatha, M. Mrunalini, Predicting the severity of bug reports using classification algorithms, in: 2016 International Conference on Circuits, Controls, Communications and Computing (I4C), 2016, pp. 1–4. doi:10.1109/CIMCA.2016.8053276.
[49]	A. F. Otoom, D. Al-Shdaifat, M. Hammad, E. E. Abdallah, Severity prediction of software bugs, in: 2016 7th International Conference on Information and Communication Systems (ICICS), 2016, pp. 92–95. doi:10.1109/IACS.2016.7476092.
[50]	K. K. Sabor, M. Hamdaqa, A. Hamou-Lhadj, Automatic prediction of the severity of bugs using stack traces, in: Proceedings of the 26th Annual International Conference on Computer Science and Software Engineering, CASCON ’16, IBM Corp., Riverton, NJ, USA, 2016, pp. 96–105.
[18]	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.
[33]	T. Zhang, J. Chen, G. Yang, B. Lee, X. Luo, Towards more accurate severity prediction and fixer recommendation of software bugs, Journal of Systems and Software 117 (2016) 166–184. doi:10.1016/j.jss.2016.02.034.
[51]	K. Jin, A. Dashbalbar, G. Yang, J.-W. Lee, B. Lee, Bug severity prediction by classifying normal bugs with text and meta-field information, Advanced Science and Technology Letters 129 (2016) 19–24. doi:10.14257/astl.2016.129.05.
[52]	T. Choeikiwong, P. Vateekul, Improve accuracy of defect severity categorization using semisupervised approach on imbalanced data sets, in: Proceedings of the International MultiConference of Engineers and Computer Scientists, Vol. 1, 2016, p..
[53]	K. Jin, A. Dashbalbar, G. Yang, B. Lee, Improving predictions about bug severity by utilizing bugs classified as normal, Contemporary Engineering Science 9 (2016) 933–942. doi:10.12988/ces.2016.6695.
[54]	K. Jin, E. C. Lee, A. Dashbalbar, J. Lee, B. Lee, Utilizing feature based classification and textual information of bug reports for severity prediction, Information 19 (2) (2016) 651–659.
[5]	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.
[55]	V. B. Singh, S. Misra, M. Sharma, Bug severity assessment in cross project context and identifying training candidates, Journal of Information and Knowledge Management 16 (01) (2017) 1750005. doi:10.1142/S0219649217500058.
[24]	N. K. S. Roy, B. Rossi, Cost-sensitive strategies for data imbalance in bug severity classification: Experimental results, in: 2017 43rd Euromicro Conference on Software Engineering and Advanced Applications (SEAA), 2017, pp. 426–429. doi:10.1109/SEAA.2017.71.

paper is written in a language other than English); EC_6 eliminated one more paper (the paper is not accessible on the Web), and another one was rejected for not satisfying IC_1. The selection phase ended up with 27 papers to be analyzed (18 from the sources plus nine from snowballing). Table 5 shows the bibliographic reference of selected papers and an reference identifier (ID) for each paper. Throughout the remainder of this text, these identifiers will be used to refer to the corresponding paper.

4.4. Categorization scheme

Petersen et al. [8] suggest the definition of a categorization scheme for conducting a systematic mapping analysis. The categories defined for this mapping study review were based on: (i) existing labels commonly used in the literature and (ii) the content of selected papers. Next sections outline the categories considered in this mapping review.

4.4.1. FLOSS software type (RQ_2)

This scheme organizes FLOSS projects by software type. Based on the taxonomy suggested by Pressman [56], FLOSS in studies belongs to three categories:

a. **Application software**: This category includes computer programs designed to solve a specific problem or business need for end users.
b. **System software**: This category includes collections of computer programs (operating systems and utilities) required to run and maintain a computer system.
c. **Programming tools**: This category includes computer programs that aid software developers in creating, debugging, maintain or perform any development-specific task.

4.4.2. Severity prediction problem (RQ_3)

This scheme drills down severity prediction problems, based on studying of selected papers, in five categories:

a. **Severe or Non-Severe (SNS)**: This category comprises the prediction problems whose the predicted severity level might be severe or non-severe. In Bugzilla, for example, the severe class would include blocker, critical, and major; and the non-severe class would include minor and trivial levels. The predictors of this problem category do not take into account the default severity level.
b. **Severe or Non-Severe With Default Class (SNSWD)**: This category is quite similar to SNS. However, the predictors consider the default severity level as severe or non-severe, or yet as another class.
c. **Multiple Classes (MC)**: This category comprises prediction problems whose predictors treat each severity level as a different class. Likewise that in SNS, the predictors do not take into account the default severity level.
d. **Multiple Classes With Default Class (MCWD)**: This category is similar to MC. However, the predictors consider the default severity level as a regular class.
e. **Blocking or Non-Blocking (BNB)**: The prediction problem whose predictors classify a bug report severity level into blocking or non-blocking class. A blocking bug is a software defect that prevents other defects from being fixed [10].

4.4.3. Feature data types (RQ_4)

This scheme groups the features used for bug reports severity prediction into five categories:

a. **Qualitative categorical**: This category encompasses features that contain an unordered list of values. For instance, bug report *platform* attribute (e.g., “Windows”, “Linux”, “Android”).
b. **Qualitative ordinal**: This category encompasses features that contain an ordered list of values. For instance, bug report *priority* attribute (e.g., P1, P2, P3).
c. **Quantitative discrete**: This category encompasses features that contain a finite or an infinite number of values that can be counted, such as the *bug fixed time* attribute.
d. **Quantitative continuous**: This category encompasses features that contain an infinite number of values that can be measured, such as the *summary weight* attribute.
e. **Unstructured text**: This category encompasses features that either do not have a pre-defined data model or are not organized in a pre-defined manner. For instance, the bug report *description* attribute usually includes free format texts.

4.4.4. Feature selection methods (RQ_5)

This scheme organizes the feature selection methods used for bug reports severity prediction into three categories, according to the taxonomy described in Guyon and Elisseeff [57]:

a. **Filter**: This category comprises methods that assign a score to the features, and use this value as a selection criterion to identify the top-scoring ones.
b. **Wrapper**: This category comprises methods that prepare, evaluate and compare feature combinations, and select the best one. Methods within this category consider the feature selection as a search problem.
c. **Embedded**: This category comprises methods that select which features best contribute to the accuracy while the ML algorithm creates the predicting model.

4.4.5. Text mining feature representations (RQ_6)

This scheme organizes the techniques for text mining feature representations used for bug report severity prediction in four categories. The following categories were employed in this mapping study review according to Feldman et al. [21]:

a. **Character**: This category includes techniques that represent features as individual component-level letters, numerals, special characters, and spaces are the building blocks of higher-level semantic features, such as words, terms, and concepts.
b. **Word**: This category includes techniques that represent features as specific words selected directly from a “native” document.
c. **Term**: This category includes techniques that represent features as single words and multiword phrases selected directly from a corpus of a native document using a term-extraction method.
d. **Concept**: This category includes techniques that represent features generated for a document employing manual, statistical, rule-based, or hybrid categorization methodologies.

4.4.6. ML algorithms categories (RQ_7)

This scheme groups ML algorithms used in bug report severity prediction in five categories. The following categories were based on taxonomy proposed by Facelli et al. [29]:

a. **Distance-based**: This category comprises algorithms that use the proximity between data to generate their predictions.
b. **Probabilistic-based**: This category comprises algorithms that make their predictions based on the Bayes’s theorem.
c. **Searching-based**: This category comprises algorithms that rely on searching for a solution in space to generate their predictions.
d. **Optimization-based**: This category comprises algorithms whose predictions are based on an optimization function.
e. **Ensemble-method**: This category comprises algorithms that combine or aggregate results of different base classifiers to make a prediction.

4.4.7. Evaluation measures categories (RQ_8)

This scheme organizes evaluation measures used in selected papers by categories. According to Japkowicz and Shah [23], there are six categories:

a. **Single class focus**: This category comprises measures based on confusion matrix information, whose focus is on a single class of interest.
b. **Multi-class focus**: This category comprises measures based on confusion matrix information, whose focus is on all the classes of the problem domain.
c. **Graphical measure**: This category comprises measures based on a confusion matrix in conjunction with extra information. They enable visualization of the classifier performance under different skew ratios and class priors distribution and are indicated for scoring classifiers.
d. **Summary statistics**: This category comprises measures based on a confusion matrix in conjunction with extra information. They enable to quantify the comparative analysis between classifiers and are indicated for scoring classifiers.
e. **Distance/error-measure**: This category comprises measures based on a confusion matrix in conjunction with extra information. They measure the distance of an instance’s predicted class label to its actual label and are recommended for continuous and probabilistic classifiers.
f. **Information theoretic measures**: This category comprises measures based on a confusion matrix in conjunction with extra information. They reward a classifier upon correct classification relative to the (typically empirical) prior to the data and are indicated for continuous and probabilistic classifiers.

4.4.8. Sampling techniques (RQ_9)

This scheme organizes the sampling techniques used for bug report severity prediction into three categories. Such categories according to Japkowicz and Shah [23] include:

a. **No re-sampling**: This category encompasses techniques that test the algorithm on a large set of unseen data.

Table 6
Papers publication sources.

Paper source	Type	Reference
ACM Symposium on Applied Computing	Conference	[9]
Advanced Science and Technology Letters & Journal	Journal	[51]
Asia-Pacific Software Engineering Conference	Conference	[25]
Computational Science and Its Applications (ICCSA)	Conference	[43]
Computer Software and Applications Conference	Conference	[35]
Contemporary Engineering Sciences	Journal	[53]
Empirical Software Engineering	Journal	[18]
Futuristic Trends on Comp. Analysis and Knowledge Mgmt. (ABLAZE)	Conference	[47]
Information and Software Technology	Journal	[46]
Intl. Conference on Circuits, Controls, Communications and Computing (I4C)	Conference	[48]
Intl. Conference on Computer Science and Software Engineering	Conference	[50]
Intl. Conference on Information and Communication Systems (ICICS)	Conference	[49]
Intl. Conference on Software Engineering and Service Science	Conference	[42]
Intl. Information Institute	Journal	[54]
Intl. Journal of Open Source Software and Process(IJOSSP)	Journal	[41]
Intl. MultiConference of Engineers and Computer Scientists	Conference	[52]
Journal of Information & Knowledge Management	Journal	[55]
Journal of Systems and Software	Journal	[33]
Procedia Computer Science	Journal	[45]
Software Engineering and Advanced Applications (SEAA)	Conference	[24,44]
Symposium on Applied Computing (SAC)	Conference	[5]
Working Conference on Mining Software Repositories (MSR)	Conference	[4,10,12,26]
Working Conference on Reverse Engineering	Conference	[3]

b. **Simple re-sampling**: This category encompasses techniques that use each data point for testing only once.
c. **Multiple re-sampling**: This category encompasses techniques that use each data point for testing more than once.

4.4.9. *Statistical tests* (RQ_{10})

This scheme groups statistical tests commonly used for bug report severity prediction in three categories. Such categories employed in this mapping study review according to Japkowicz and Shah [23] include:

a. **Parametric**: This category comprises methods that make strong assumptions about the distribution of the population
b. **Non-parametric**: This category comprises methods that do not make strong assumptions about the distribution of the population
c. **Parametric and non-parametric**: This category comprises methods that are both parametric and non-parametric

4.4.10. *Experiment software tools* (RQ_{11})

This scheme groups the software tools used in experiments for bug reports severity prediction in two well-known and popular categories:

a. **Free/Libre Open Source Software (FLOSS)**: This category includes software that can be freely used, modified, and redistributed.
b. **Closed Source Software (CSS)**: This category includes software that is owned by an individual or a company whose source code is not shared with the public for anyone to look at or change.

5. Results

This section summarizes the results of this mapping study review. The answer to each research question (from RQ1 to RQ12) is presented in tables and charts. The organization of extracted data follows the criteria defined in Section 4.

5.1. *When and where have the studies been published? (RQ1)*

Fig. 4a shows that research on bug report severity prediction in FLOSS is recent and active with a vast number of papers (22 out of 27 ≈ 81%) published from 2014. Journals and conferences on information technology, mining software repositories, and software engineering seem to be more open to papers of bug report severity prediction (Table 6).

Table 7
Paper distribution by FLOSS.

Project	References	Total
Eclipse	[3–5,9,10,12,18,24–26,33,35,41,42,44–47,49–55]	25
Mozilla	[3,4,9,10,18,24,25,33,35,41,43,44,46,48,49,51,53–55]	19
Openoffice	[3,10,18,33,46]	5
Netbeans	[10,33,35,46]	4
Gnome	[4,12,41]	3
Chromium	[10,46]	2
Freedesktop	[10,46]	2
GCC	[33]	1
Hibernate	[24]	1
Spring	[24]	1
Android	[5]	1
JBoss	[5]	1
Mongodb	[24]	1
WineHQ	[48]	1

5.2. *What FLOSS are the most used as experimental target for bug report severity prediction (RQ2)?*

Table 7 shows that most papers concentrated their focus on five FLOSS: Eclipse (≈ 92%), Mozilla (≈ 70%), Openoffice (≈ 18%), Netbeans (≈ 14%), and Gnome (≈ 11%). The detailing of each FLOSS including description, category, BTS, and URL is presented in Table 8.

Fig. 4b shows that most papers worked with FLOSS related to Programming Tool (≈ 92%) and Application Software (≈ 74%) categories. Moreover, it presents that 17 out of 27 (≈ 62%) papers worked with both categories. Fig. 4c highlights that all papers handled bug reports extracted from Bugzilla, and a just few papers from Google (3 out of 27 ≈ 11%) and Jira (2 out of 27 ≈ 7%). Only one paper (Yang et al. [5]) investigated bug reports from three BTS.

5.3. *Was bug report severity prediction most addressed as either a fine-grained label or coarse-grained label prediction problem (RQ3)?*

Table 9 shows that about the same fraction of papers addressed the bug report severity prediction as coarse-grained and fine-grained problem. Fig. 5a shows that most papers (10 out of 27 ≈ 37%) addressed the severity prediction as a problem of SNS category. Moreover, 8 out of 27 (≈ 20%) papers addressed it as a problem of the MC category and 5 out of 27 (≈ 18.5%) addressed a problem of the MCWD category. A few

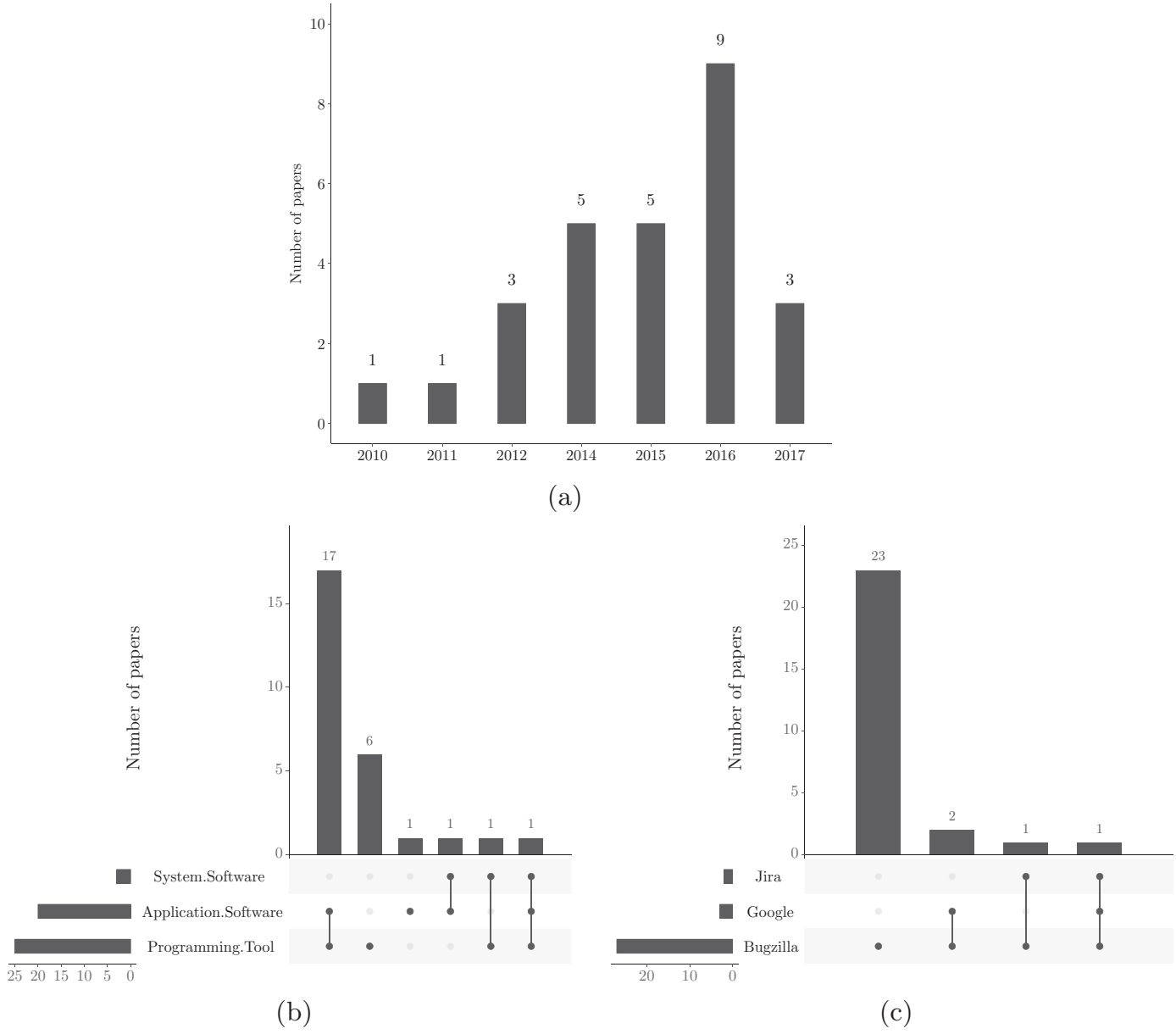

Fig. 4. (a) Paper distribution by year (67% published in conferences and 33% in journals), (b) paper distribution by FLOSS (c) paper distribution by BTS.

Table 8
FLOSS investigated in papers.

Project	Description	Category	BTS	URL (As of 2019/07/31 07:16:16)
Android	Operating syst.	Syst. software	Google	https://issuetracker.google.com
Chromium	Web browser	App. software	Google	https://www.chromium.org/issue-tracking
Eclipse	IDE	Prog. tool	Bugzilla	https://bugs.eclipse.org/bugs/
FreeDesktop	Desktop software	Prog. tool	Bugzilla	https://bugs.freedesktop.org/
GCC	C compiler	Prog. tool	Bugzilla	https://gcc.gnu.org/bugzilla/
Gnome	X Windows Env.	App. software	Bugzilla	https://gcc.gnu.org/bugzilla/
Hibernate	ORM framework	Prog. tool	Jira	https://hibernate.atlassian.net
Jboss	App. server	Syst. software	Jira	https://issues.jboss.org/s
Mongodb	Nosql database	Syst. software	Jira	https://jira.mongodb.org/
Mozilla	Internet tools	App. software	Bugzilla	https://bugzilla.mozilla.org/
Netbeans	IDE	Prog. tool	Bugzilla	https://netbeans.org/bugzilla/
OpenOffice	Office suite	App. software	Bugzilla	https://bz.apache.org/ooo/
Spring	JEE framework	Prog. tool	Jira	https://jira.spring.io
WineHQ	Compatibility layer	Syst. software	Bugzilla	https://bugs.winehq.org/

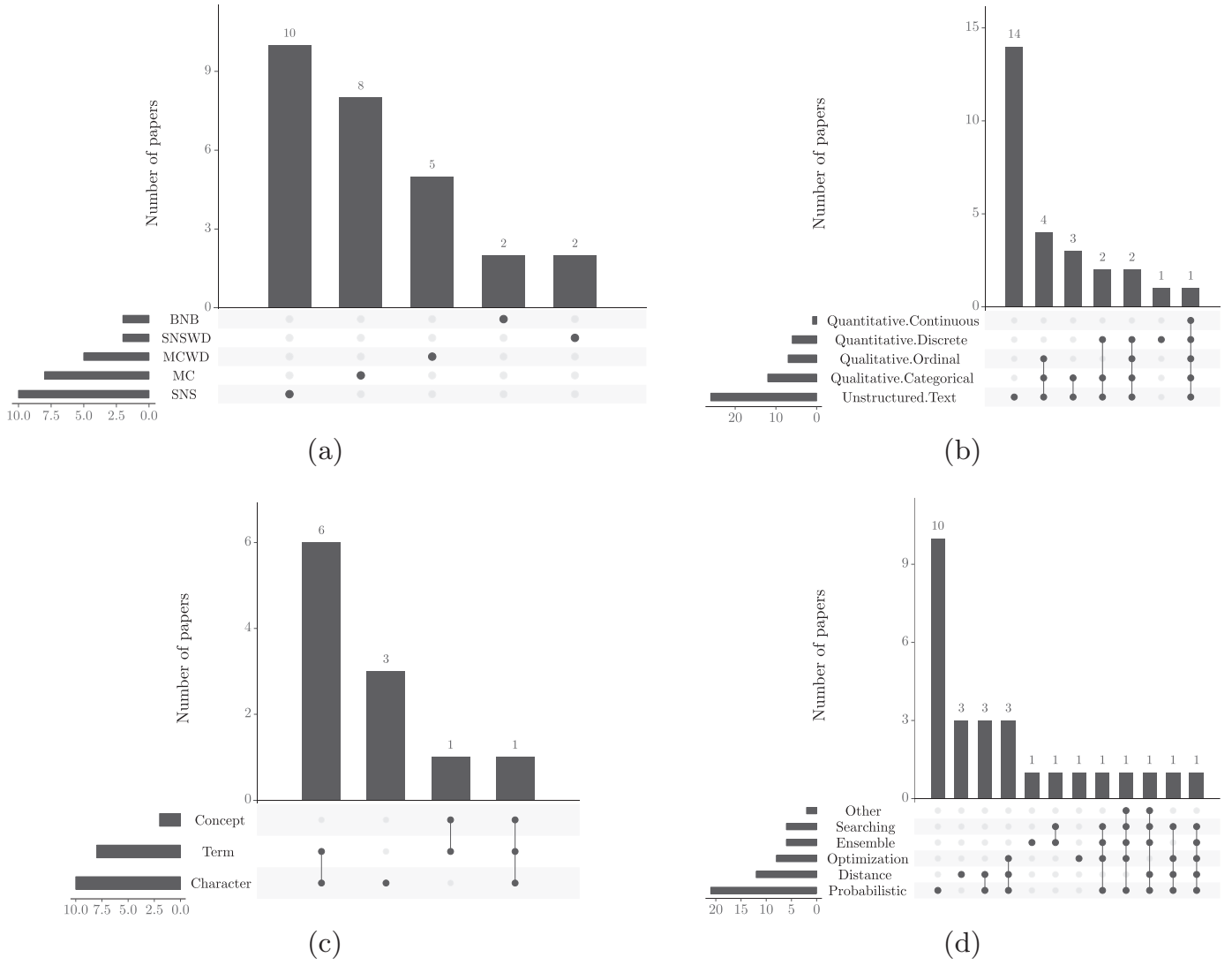

Fig. 5. (a) Paper distribution by prediction problem category, (b) paper distribution by feature data category, (c) paper distribution by text mining method category (d) paper distribution by ML algorithm categories.

Table 9
Paper distribution by prediction problem.

Problem	References	Total
Coarse-grained	[4,10,12,25,26,42,44–47,49,51,53,54]	14
Fine-grained	[3,5,9,18,24,33,35,41,43,48,50,52,55]	13

papers addressed a BNB (2 out of 27 ≈ 7%) or an SNSWD (2 out of 27 ≈ 7%) problem. No paper addressed more than one problem category.

5.4. What are the most common features used for bug report severity prediction (RQ4)?

Table 10 describes the features used for bug report severity prediction. Each feature in this table has a description, a category, an indicator of if the feature is computed from others, and an indicator of if the feature is available in Bugzilla, Jira, and Google.

As shown in Table 11, most used features were *summary* (≈ 74%) and *description* (≈ 70.37%). Besides, a significant number of papers also reported using the features *product* (≈ 37%) and *component* (≈ 33%).

Regarding data type of the features, unstructured text (26 out of 27 ≈ 96%) was most used data type in the papers (Fig. 5b). Only one paper (Meera et al. [43]) reported the use of features belonging to each data type presented in the chart.

5.5. What are the most common features selection methods employed for bug report severity prediction (RQ5)?

Table 12 shows that few papers (≈ 40%) use of a feature selection method for bug report severity prediction and remarkably all papers only used filter category methods.

5.6. What are the most used text mining methods for bug report severity prediction (RQ6)?

As shown in Table 13, ≈ 74% out of the papers used unigrams for feature vector extraction and ≈ 33% out of the papers used TF-IDF for feature vector weighting. Regarding similarity/dissimilarity functions, ≈ 7% reported using some function: one paper used BM25, one paper BM25ext, and another KL divergence. Fig. 5c shows that most used text mining method (9 out of 15 ≈ 44%) belongs to term category. Besides,

Table 10
Feature descriptions.

Feature	Description	Category	Comp.	Bugz.	Jira	Google
Attachments	Files (e.g., test cases or patches) attached to bugs.	Qualitative Categorical	No	Yes	No	Yes
Bug fix time	Time to fix a bug (Last Resolved Time - Opened Time).	Quantitative Continuous	Yes	Yes	Yes	Yes
Bug id	Bug report identifier.	Quantitative Discrete	No	Yes	Yes	Yes
CC list	A list of people who get mail when the bugs changes.	Qualitative Categorical	No	Yes	No	Yes
Comment	Textual content appearing in the comments of bug report.	Unstructured Text	Yes	Yes	Yes	Yes
Comment size	The number of word of all comments of a bug.	Quantitative Discrete	Yes	Yes	Yes	Yes
Complexity	Bug level of complexity based on bug fix time.	Qualitative Ordinal	Yes	Yes	Yes	Yes
Component	Each product is divided into different components (e.g., Core, Editor, and UI).	Qualitative Categorical	No	Yes	Yes	Yes
Description	Textual content appearing in the description field of the bug report.	Unstructured Text	No	Yes	Yes	Yes
Description size	The number of words in the description.	Quantitative Discrete	Yes	Yes	Yes	Yes
Importance	The importance of a bug is a combination of its priority and severity.	Qualitative Discrete	No	Yes	No	No
Number in CC list	The number of developers in the CC list of the bug.	Quantitative Discrete	Yes	Yes	No	Yes
Number of comments	Number of comments added to a bug by users.	Quantitative Discrete	Yes	Yes	Yes	Yes
Number of dependents	Number of dependents of a bug report.	Quantitative Discrete	Yes	Yes	Yes	Yes
Number of duplicates	Number of duplicates of a bug report.	Quantitative Discrete	Yes	No	No	Yes
Platform	Indicates the computing environment where the bug was found (e.g., Windows, GNU/Linux, and Android).	Qualitative Categorical	No	Yes	Yes	Yes
Priority	Priority should normally be set by the managers, maintainers or developers who plan to work, not by the one filling the bug or by outside observers.	Qualitative Ordinal	No	Yes	Yes	Yes
Product	What general “area” the bug belongs to (e.g., Firefox, Thunderbird, and Mailer).	Qualitative Categorical	No	Yes	Yes[a]	No
Report length	The content length of long description providing debugging information.	Quantitative Discrete	Yes	Yes	Yes	Yes
Reporter	The account of the user who created the bug report.	Qualitative Categorical	No	Yes	Yes	Yes
Reporter blocking experience	Counts the number of blocking bugs filed by reporter previous to this bug.	Quantitative Discrete	Yes	Yes	Yes	No
Reporter experience	Counts the number of previous bug reports filed by the reporter.	Quantitative Discrete	Yes	Yes	Yes	No
Reporter name	Name of the developer or user that files the bug	Qualitative Categorical	No	Yes	Yes	No
Severity	Indicates how severe the problem is - from blocker (“application unusable”) to trivial (“minor cosmetic issue”).	Qualitative Ordinal	No	Yes	Yes	No
Summary	A one-sentence summary of the problem.	Unstructured Text	No	Yes	Yes	Yes
Summary weight	Score calculated using the information gain criterion.	Quantitative Continuous	Yes	Yes	Yes	Yes

[a] In Jira, *product* corresponds to *project* field.

only one paper (Yang et al. [5]) reported the use of text mining belonging to each category presented in the chart.

5.7. What are the most used machine learning algorithms for bug report severity prediction (RQ7)?

Table 14 presents that most papers used KNN and NB ($\approx$ 44%) and that NBM has also been used quite frequently ($\approx$ 40%). As shown in Fig. 5d, the majority of the papers used a probabilistic-based algorithm (21 out of 27 $\approx$ 77%). The figure also shows that 15 out of 27 papers ($\approx$ 55%) used algorithms belonging to a ML algorithm single category.

5.8. What are the measures used to evaluate ML algorithms performance for bug report severity prediction (RQ8)?

Most papers evaluated the ML model accuracy in bug report severity prediction using three measures based on the confusion matrix (Table 15): precision ($\approx$ 77%), recall ($\approx$ 70%), and f-measure ($\approx$ 66%). The majority of the used measures are of the single class category (21 out of 27 $\approx$ 70%), as shown in Fig. 6a. Only one paper (Roy et al. [44]) investigated all four categories of measures presented in the chart.

5.9. Which sampling techniques are applied most frequently to generate more reliable predictive performance estimates in severity prediction of a bug report (RQ9)?

Most papers shown in Table 16 applied 10-Fold CV ($\approx$ 66%) to generate more reliable effectiveness estimates. Fig. 6b displays that the majority of those papers (14 out of 15 $\approx$ 93%) employed a simple resampling method. Also, only one paper (Otoom et al. [49]) investigated both non-resampling and simple resampling methods for bug report severity prediction.

5.10. Which statistical tests were used to compare the performance between two or more ML algorithms for bug report severity prediction (RQ10)?

As shown in Table 17, most of the papers used Wilcoxon Signed Rank Test (8 out of 27 $\approx$ 29%) and *t*-test (7 out of 27 - $\approx$ 26%) to compare the performance of ML algorithms. Fig. 6c calls attention that only ten papers ($\approx$ 37%) reported using any statistical test in their experiments to bug report severity prediction. The figure also shows that most of these papers (8 out of 10 $\approx$ 80%) applied a non-parametric test, and 5

Table 11
Paper distribution by features.

Feature	References	Total
Summary	[3–5,12,18,24–26,33,35,41–43,45,48,49,51,53–55]	20
Description	[3–5,9,10,18,24,33,35,42,44,46–51,53,54]	19
Product	[3,10,18,33,35,46,48,51,53,54]	10
Component	[3,10,33,35,46,48,51,53,54]	9
Severity	[10,46,51,53,54]	5
Priority	[10,35,43,46]	4
Reporter	[51,53,54]	3
Description size	[10,46]	2
Number in CC list	[10,46]	2
Reporter blocking experience	[10,46]	2
Reporter experience	[10,46]	2
Reporter name	[10,46]	2
Attachments	[42]	1
Bug fix time	[43]	1
Bug Id	[52]	1
CC list	[43]	1
Comment	[10]	1
Comment size	[46]	1
Complexity	[43]	1
Importance	[48]	1
Number of comments	[43]	1
Number of dependents	[43]	1
Number of duplicates	[43]	1
Platform	[46]	1
Report length	[42]	1
Summary weight	[43]	1

Table 12
Paper distribution by feature selection methods.

Method	Category	References	Total
Information Gain	Filter	[18,24,25,41,45]	5
Chi-Square	Filter	[25,44,45]	3
Correlation Coefficient	Filter	[25]	1

out of 10 (50%) papers employed both parametric and non-parametric tests.

5.11. Which software tools were used to run bug report severity prediction experiments (RQ11)?

Most of the papers listed in Table 18 used RapidMiner (5 out of 18 ≈ 27%) to run their experiments. However, Fig. 6d presents a strong predominance of FLOSS over CSS, 13 out of 18 papers (≈ 61%) executed FLOSS to perform their experiments. The figure also shows that no paper used both CSS and FLOSS.

5.12. Which solution types were proposed for bug report severity prediction problem (RQ12)?

Most of the papers (26 out of 27 ≈ 93%) provided offline solutions (Table 19). Just one paper (Tian et al. [3]) presented an online solution.

5.13. ML algorithms performance summary

The evaluation of ML algorithms performance, both in absolute terms and in relation to other algorithms, involves performance measures, statistical significance testing, sampling techniques and error estimation, and test benchmark selection [23]. This evaluation is particularity difficult in our context because the authors of selected studies have used different datasets, metrics, features, and even different software tools, which complicate the fair evaluation of existing methods. Table 20 provides a quantitative view of the performance obtained in the experiments, showing the algorithms which presented the best performances in each paper. This table includes paper reference, open source project considered, ML Algorithm, hyper-parameters, number of classes, three common measures adopted (e.g., AUC, Accuracy, F-Measure), and the number of terms used in each paper. It is worth to call attention

Table 13
Paper distribution by text mining models.

Technique	Category	References	Total
Vector Extraction Models			
Unigrams	Character	[3–5,12,18,24,25,33,35,41,42,44,45,47,49–55]	20
Bigrams	Character	[3,18,33,44]	4
Topic model	Concept	[9,33,35]	3
Vector Weighting Models			
TF-IDF	Term	[9,12,24,41,44,45,47,50,55]	9
TF	Term	[12,25,49]	3
Binary	Term	[12]	1
Vector Similarity/Dissimilarity Functions			
BM25ext	Term	[33]	1
BM25	Term	[3]	1
KL divergence	Term	[35]	1

Table 14
Papers distribution by ML algorithms.

Algorithm	Category	References	Total
KNN	Distance	[3,9,10,12,18,33,35,43,47,50,52,55]	12
NB	Probabilistic	[4,9,10,12,26,35,41,43,44,49,52,55]	12
MNB	Probabilistic	[5,9,12,18,25,42,45,47,51,53,54]	11
SVM	Otimization	[12,18,24,41,43,52,55]	7
Random Forest	Ensemble	[10,41,46,49,52]	5
C4.5[a]	Searching	[10,41,48]	3
Bagging[b]	Ensemble	[46,48]	2
Decision Tree	Searching	[18,52]	2
AdaBoost[c]	Ensemble	[49]	1
Functional Tree[d]	Searching	[49]	1
Random Tree[e]	Ensemble	[49]	1
RBF Networks[f]	Optimization	[49]	1
RIPPER[g]	Other	[41]	1
Zero-R[h]	Other	[10]	1

[a] C4.5 [10] is an algorithm used to generate a decision tree which follows a greedy divide and conquer strategy in training step.
[b] Bagging Ensemble [46] uses multiple learning algorithms to obtain better predictive performance that could be received from any of the constituent learning algorithms alone. This approach involves having each model in the ensemble vote with equal weight. To promote model variance, bagging trains each model in the ensemble using a randomly drawn subset of the training set.
[c] AdaBoost [49] is an ensemble algorithm that attempts to produce a very accurate classification rule by combining moderately inaccurate weak classifiers.
[d] Functional Tree [49] is a classification tree that could have logistic regression function at the inner nodes and or leaves.
[e] Random Tree [49] is an ensemble classifier that consists of directed graphs build with a random process.
[f] Radial Basis Function (RBF) Neural Network [14] is a neural network that consists of input nodes connected by weights to a set of RBF neuron, which fire proportionally to the distance between the input and the neuron in weight space. RBF is a real-valued function whose value depends only on the distance from the origin.
[g] RIPPER [58] is a rule-based classifier that builds a set of rules that identify the classes while minimizing the amount of error. The error is defined by the number of training examples misclassified by the regulations.
[h] Zero-R [10] is the simplest classifier which always predicts the majority class in training set. It is commonly employed for determining a baseline performance as a benchmark for other methods.

Table 15
Papers distribution by evaluation measures.

Measure	Category	References	Total
Precision	Single Class	[3–5,9,10,24,33,35,41,43–48,50–55]	21
Recall	Single Class	[3–5,9,10,24,33,35,41,43,44,46,48,50–55]	19
F-measure	Single Class	[3–5,9,10,24,33,35,41,43,44,46,50–55]	18
Accuracy	Multi-Class	[10,26,35,41,43–45,47–49,55]	11
AUC	Summary	[4,12,25,42,44]	5
ROC	Graphical	[4,44]	2
MRR[a]	Single Class	[33,35]	2
Effectiveness Ratio[b]	Single Class	[46]	1
Krippendorff's Alpha Reliability[c]	Single Class	[18]	1

[a] Mean Reciprocal Rank (MRR) [17] is the average of reciprocal ranks of results of a set of questions. A reciprocal rank of a question is the multiplicative inverse of the rank of the first correct answer.
[b] Effectiveness Ratio [46] is a cost-effectiveness measure, which evaluates prediction performance given a cost limit.
[c] Krippendorff's Alpha Coefficient [18] is a statistical measure of the agreement achieved when coding a set o units analysis of values of a variable.

Table 16
Paper distribution by sampling methods.

Method	Category	References	Total
10-Fold CV	Simple Re-sampling	[5,12,24,25,41,42,46,48,49,53]	10
Stratified 10-Fold CV[a]	Simple Re-sampling	[10,44,55]	3
SMOTE[b]	Simple Re-sampling	[46,52]	2
Hold-out	No Re-sampling	[49]	1
03-Fold CV	Simple Re-sampling	[52]	1
05-Fold CV	Simple Re-sampling	[45]	1

[a] Stratified k-fold CV [23] is a k-fold CV variation, which ensures that class distribution is respected in the training and testing sets created at every fold.
[b] Synthetic Minority Oversampling TEchnique (SMOTE) [59] is an sampling approach in which the minority class is over-sampled by creating "synthetic" examples rather than by over-sampling with replacement.

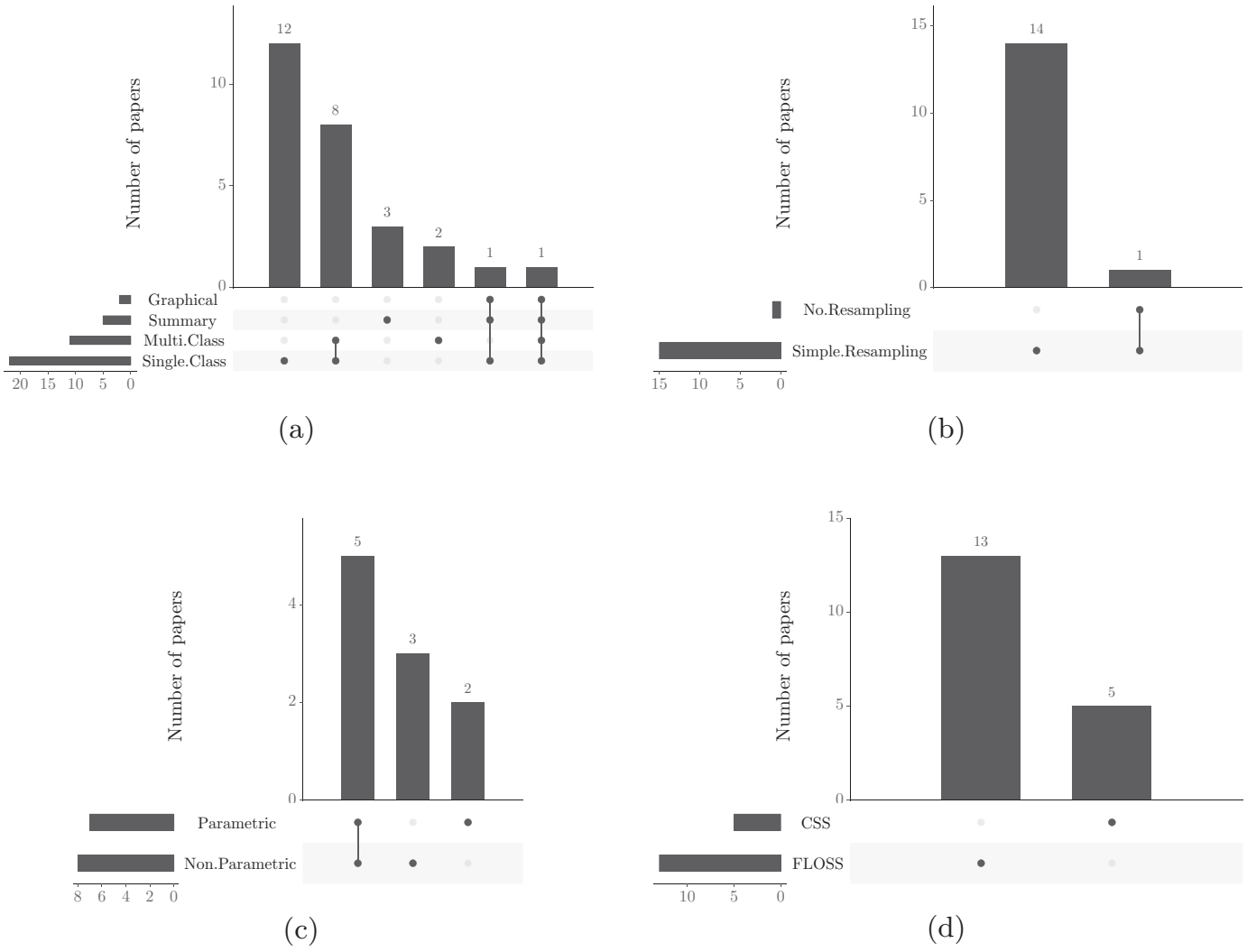

Fig. 6. (a) Paper distribution by evaluate measure categories, (b) paper distribution by sampling method categories, (c) paper distribution by statistical test categories, and (d) paper distribution by software tool categories.

Table 17
Paper distribution by statistical tests.

Test	Category	References	Total
Wilcoxon signed rank test	Non-parametric	[5,9,10,33,35,46,53,55]	8
T-test	Parametric	[5,9,24,33,35,52,53]	7
Proportion test	Parametric	[24]	1
Shapiro-Wilk test	Parametric	[5]	1

Table 18
Paper distribution by software tools.

Tool	Category	References	Total
RapidMiner	CSS	[41,43,45,47,55]	5
WEKA	FLOSS	[12,18,46,48]	4
R	FLOSS	[5,9,35]	3
NLTK	FLOSS	[33,44,53]	3
Statisca	CSS	[41,43]	2
TMT	FLOSS	[33,35]	2
OpenNLP	FLOSS	[3]	1
WordNet	FLOSS	[44]	1
Ruby	FLOSS	[4]	1
WVTool	FLOSS	[25]	1
CoreNLP	FLOSS	[5]	1

Table 19
Solution types proposed in selected papers.

Tool	References	Total
offline	[4,5,9,10,12,18,24–26,33,35,41–55]	26
online	[3]	1

to the average scores of measures presented in the papers for the most measures, which are shown in the table.

6. Discussion

This section presents lessons learned from this systematic mapping study and recommends possible research directions for the bug severity prediction problem.

Table 20
The best ML algorithms performance of each selected paper.

Ref.	Projects	Algorithms	Hyper parameters	Classes	Metrics			Terms
					AUC	Accuracy	F-Measure	
[4]	Gnome	NB		2	0.869		0.810	20
[12]	Gnome	MNB		2	0.930			60
[3]	Openoffice	KNN	$K = 1$	5			0.430	
[25]	Mozilla	MNB		2	0.849			100
[41]	Eclipse	SVM		5		0.609		25
[35]	Mozilla	KNN	$K = 1$	7		0.750		40
[42]	Eclipse	MNB		2	0.764			
[10]	Mozilla	Random Forest		2		0.823	0.432	
[43]	Mozilla	NB		6		0.916		
[44]	Mozilla	NB		2	0.924	0.880	0.797	750
[26]	Eclipse	NB		3		0.490		
[9]	Eclipse	Concept Profile		6			0.848	
[45]	Eclipse	KNN		2		0.915		125
[46]	Mozilla	Bagging Ensemble		2			0.482	
[47]	Eclipse	MNB		2		0.720		
[48]	Mozilla	Bagging Ensemble		4		0.812		
[49]	Eclipse/ Mozilla	Random Forest		2		0.764		59
[50]	Eclipse	KNN	$K = 10$	5			0.330	
[33]	Eclipse	KNN	$K = 5$	5			0.380	30
[51]	Mozilla	MNB		2			0.820	79,000
[52]	Eclipse	OS-YATSI (KNN based)		3			0.807	
[53]	Mozilla	MNB		2			0.820	
[54]	Mozilla	MNB		2			0.790	
[5]	Android	MNB		5			0.716	142,359
[55]	Eclipse	KNN		5			0.615	100
[24]	Many(*)	SVM		5			0.950	

Researchers have concentrated their focus on few FLOSS, namely Eclipse and Mozilla, both hosted by Bugzilla. The main reason given by most researchers is benchmarking their results with Lamkanfi et al. [4,12], which are often considered the state-of-the-art in this research area. Notably, they have not considered relevant FLOSS projects like as Linux Kernel, Spark, MySQL, among others. Moreover, published research only marginally investigated two other bug trackers mentioned in reviewed papers: Jira and Google Issue Tracker. The first one tracks bugs for more than 80% Apache Foundation's projects[4], and the last one tracks bugs for many internal and external Google Inc. projects[5] Surprisingly, for example, no paper investigated Github, another outstanding collaborative website which hosted more than 67 million projects (including many FLOSS)[6].

From 14 FLOSS identified in this mapping study review, six are programming tools, four are system software, and four are application software. For former two, people who report bugs are typically technical users, and, for the latter, people are end-users. A bug report can be more or less detailed depending on who reports it [4,12]. Therefore, the probability of writing more accurate bug reports is larger for technical users than end-users.

Researchers considered the bug report severity prediction using five different approaches: three of them (SNS, SNSWB, and BNB) predict the severity level from two or three class (coarse-grained). Two other approaches (MC and MCWD) seem to be more complex than problems of the previous ones because the severity level may be predicted from five or six classes (fine-grained).

From another aspect, two categories (SNSWD and MCWD) did not consider the default severity level (in Bugzilla, often "normal"). By doing so, they have argued that they would avoid noisy and unreliable data. Saha et al.[26] confirmed that many bug reports in practice are not "normal" and this misclassification can really affect the accuracy of ML algorithms.

Most of the approaches proposed by researchers to predict bug report severity relied on unstructured text features (*summary* and *description*). Because of this, text mining techniques played, side by side with machine learning methods, an essential role in delivering appropriated solutions. Two other features (*product* and *component*) was also quite used in these approaches. That may indicate that bug reports collected from separate parts of a single FLOSS yield different ML algorithms outcomes.

Regarding available feature data types in bug reports, more than half of them are qualitative. SVM and Neural Networks, two popular and essential ML algorithms in modern machine learning [14], do not work with qualitative data [13]. These two facts bring an extra challenge to use qualitative features for bug report severity prediction. When the input data set has qualitative data, categorical, and ordinal values must be converted into numeric values before applying ML algorithms.

Despite a large number of features raised by the application of text mining activities (e.g., tokenizing, stop word removal and stemming) on *summary* and *description* and of its effectiveness in SNS and SNSWD problems [25], few papers employed feature selection methods for bug severity prediction. Furthermore, these papers notably only investigated filter feature selection methods. Filter-based approaches have two drawbacks [13]: (i) they do not take into account redundancy between features, and (ii) they do not detect dependencies between them.

Most papers reported the use of unigram for feature extraction. However, few papers explicitly report the use of another text mining methods. Among these papers, the most used method feature vector weighting was TF-IDF and the most used method for verifying text similarity was BM25.

The number of terms extracted by text mining methods is considered relevant to predict bug severity because this number can affect the overall performance of ML algorithms and, more specifically, the model accuracy. As shown in Table 20, the most frequent number of terms used in these papers ranges from 25 to 125. However, the table also shows that many surveyed papers did not mention the number of terms used in experiments.

It seems that researchers have adopted a conservative approach regarding the use of ML algorithms. Most papers applied at least one of

[4] http://www.apache.org/index.html#projects-list (As of september 2018).
[5] https://developers.google.com/issue-tracker/ (As of september 2018).
[6] https://octoverse.github.com/ (As of september 2018).

the following well-known and traditional supervised algorithms: k-NN, Näive Bayes, and its extension Näive Bayes Multinomial.

Researchers evaluated the performance of ML algorithms in their proposed approaches using precision, recall, and f-Measure. Three common measures employed in ML arena, but which may strongly skew in the imbalanced scenario [60,79]. This characteristic is particularly critical in bug report repositories which are intrinsically imbalanced in practice [24]. To minimize such distortion, Tian et al. [18], instead, recommend measuring performance using inter-rater agreement based metrics, such as Cohen's Kappa [61] or Krippendorff's Alpha [62]. Despite its well-known issues in an imbalanced dataset [23], quite few papers still used accuracy to measure ML performance. AUC, an alternative used by few papers, seems more robust than accuracy in imbalanced data conditions. Like Kappa and Krippendorff's Alpha, AUC considers class distribution on performance evaluation of ML algorithms for bug report severity prediction.

In most of surveyed papers, no clear description is provided on how hyper-parameters were tuned. For example, in papers [3,33,55], no details regarding parameter tuning is discussed. In some cases, even values used to hyper-parameters are not properly presented [10,24,49].

Few papers reported the use of a resampling approach to improving the accuracy of ML algorithms. The papers only reported the use of simple resampling strategies, including, the most used, k-fold cross-validation. Only one paper used SMOTE, considered a "de facto" resampling method in imbalanced data scenario [63]. Japkowicz et el. [23] also suggest that experiments may achieve better results using multiple resampling methods, for example, repeated k-fold cross-validation.

It seems that most researchers compared the results yielded in their experiments with others observing only the means of measures applied. Few of them reported utilizing statistical tests to do that, which is a recommended practice in empirical software engineering [64]. Most used statistical test was parametric. This kind of test requires that samples follow a statistical distribution (e.g., normal), which is not guaranteed to occur in practice when comparing ML models [29]. Besides, no paper investigated Analysis of Variance (ANOVA), a robust ranked-based test recommended by Kitchenham et al. [64] for Empirical Software Engineering.

Researchers preferred to use FLOSS tools to perform their analyses. Although, many papers have used a proprietary software named RapidMiner. Interestingly, researchers demonstrated a low interest in top-ranked ML tools based on R and Python programming languages.

It seems that most proposed solutions for bug report severity prediction were conceived to run in offline mode, just one paper claimed which the published solution is mature and fast enough to be embedded, for example, in a web browser. The need for a high number of labeled bug reports during the training phase [65] is a problem intrinsic to supervised ML algorithms, which may make it difficult to turn these offline solutions to online.

7. Threats of the validity of this survey

The main threats to validity for this mapping review are summarized below:

- **Selection of Journals and Conferences**: The current mapping review surveyed 23 journals and conferences considered to be leaders in software engineering in general and empirical software engineering in particular. Nevertheless, a mapping review that includes other documents such as theses, technical reports, and working papers, describing controlled experiments in bug report severity prediction would, in principle, provide more data and allow a more general conclusion to be drawn [7].
- **Selection of Papers**: To help ensure an unbiased selection process, the current mapping review defined research questions in advance, organized the selection of papers a multistage process, engaged three researchers in this process, and documented the reasons for inclusion/exclusion as suggested in [7]. Alternative procedures for paper selection may lead to a different corpus for analysis.
- **Mapping review completeness can never be guaranteed** [39]: Although this mapping review has observed a strict research protocol to ensure the relatively whole population of the relevant literature, some important papers might be missed.
- **Terminology problems in search string may lead to miss some primary studies** [40]: The terminology applied in the database query is normally accepted and used within the scientific community. Nevertheless, different terms may be used to retrieve the same relevant information.
- **Subjective evaluation of inclusion and exclusion criteria may cause misinterpretation** [39]: Explicit criteria have been defined for assessing the relevance of selected papers (Section 4.2). However, the evaluation was based on the perception and experience of the authors. Different people may have other views regarding the relevance of these papers.
- **Data Extraction**: The data were extracted from the papers by one researcher and revised by two others. Data extraction from prose is difficult [39] at the origin and the lack of standard terminology and standards for reporting experiments in software engineering may have resulted in some inaccuracy in the data.
- **Classification to Categories**: The category classification was difficult due to the lack of a well-defined method for classifying according to the schemes used. To mitigate possible misleading, this process was double-checked by two researchers.

8. Conclusions and research directions

A mapping study review provides a structure for a research report type, which enables categorizing and giving a visual summary of results that have been published in papers of a research area[8]. This map aids to identify gaps in a research area, becoming a basis to guide new research activities[7]. The current mapping review captured the current state of research on bug report severity prediction, characterized related problems and identified the main approaches employed to solve them. These objectives were reached by conducting a mapping of existing literature. In total, the review identified 27 relevant papers and analyzed them along 12 dimensions. Although these papers have made valuable contributions in bug report severity prediction, the panorama presented in this mapping study review suggests that there are potential research opportunities for further improvements in this topic. Among them, the following research directions appear to be more promising:

- There is an apparent lack of investigation on bug report severity prediction in other relevant FLOSS such as, for example, Linux Kernel, Ubuntu Linux, and MySQL, and in others BTS, for example, Github.
- Often, technical users report most bugs. Thus, the influence of user experience in predicting outcomes is still overlooked.
- Bug reports labeled with default severity level (often "normal") were prevalent in the most datasets used in reviewed papers. However, they are considered unreliable [26], and just discarding them also does not seem appropriate. Then, efforts in researching on novel approaches to handle this type of report should be considered to improve the state-of-the-art of severity prediction algorithms.
- Most approaches were based on unstructured text features (*summary* and *description*). To handle them, researchers chose to use the traditional bag-of-words approach instead of more recent text mining methods (e.g., word-embedding [66]) or data-driven feature engineering methods which may likely improve outcomes yielded so far.
- There is a clear research opportunity to investigate whether state-of-the-art ML algorithms might outperform the traditional algorithms used in all reviewed papers for bug report severity prediction. The investigation of the use of Deep learning algorithms which perform

very well when classifying audio, text, and image data [67] seems to be a promising research direction.
- Researchers should investigate more recent techniques (e.g., continuous learning [68]) to provide an approach for bug report prediction which could be employed in real-world scenarios.
- Many bug reports are resolved in a few days (or in a few hours) [69]. Efforts to predict severity level for these group of bug reports do not seem very useful. Thus, an investigation to confirm this hypothesis and to determine when the severity prediction is more appropriate in bug report lifecycle is of critical importance.
- The primary objective of our mapping study was to review the research approaches for severity level prediction in FLOSS. However, it would be a promising research venue to extend this study to commercial systems, and verify whether the same findings apply to these systems.
- The approaches published in selected papers for severity level prediction did not consider the temporal information changes in bug report features. Investigating the impact of the temporal evolution of a bug report information [70] in severity level prediction accuracy, as well as investigating data structures to store the representation of this temporal evolution seems a relevant research venue.
- Related research areas, such as severity level prediction in help desk systems for IT service management [71], may take advantage of the best practices used in severity level prediction in FLOSS. Assessing these best practices in this context is also a promising research venue.

Declaration of Competing Interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

Acknowlgedgments

Authors are grateful to CAPES (grant #88881.145912/2017-01), CNPq (grant #307560/2016-3), FAPESP (grants #2014/12236-1, #2015/24494-8, #2016/50250-1, and #2017/20945-0), the FAPESP-Microsoft Virtual Institute (grants #2013/50155-0, #2013/50169-1, 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.d.M.S. 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, J. Softw. 26 (7) (2014) 620–653, doi:10.1002/smr.1639.
[2] I. Sommerville, Software engineering, 2010, doi:10.1111/j.1365-2362.2005.01463.x.
[3] 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:10.1109/WCRE.2012.31.
[4] 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.
[5] 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.
[6] J. Uddin, R. Ghazali, M.M. Deris, R. Naseem, H. Shah, A survey on bug prioritization, Artif. Intell. Rev. 47 (2) (2017) 145–180, doi:10.1007/s10462-016-9478-6.
[7] B. Kitchenham, S. Charters, Guidelines for performing systematic literature reviews in software engineering, 2007.
[8] K. Petersen, R. Feldt, S. Mujtaba, M. Mattsson, Systematic mapping studies in software engineering, in: Proceedings of the 12th International Conference on Evaluation and Assessment in Software Engineering, EASE'08, BCS Learning & Development Ltd., Swindon, UK, 2008, pp. 68–77.
[9] 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.
[10] H.V. 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.
[11] T. Zimmermann, R. Premraj, N. Bettenburg, S. Just, A. Schroter, C. Weiss, What makes a good bug report?, 2010, 36, 618–643, doi:10.1109/TSE.2010.63.
[12] 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.
[13] P. Flach, Machine Learning: The Art and Science of Algorithms That Make Sense of Data, Cambridge University Press, New York, NY, USA, 2012.
[14] S. Marsland, Machine Learning: An Algorithmic Perspective, second ed., Chapman & Hall/CRC, 2014.
[15] S.B. Kim, K.S. Han, H.C. Rim, S.H. Myaeng, Some effective techniques for naive bayes text classification, IEEE Trans. Knowl. Data Eng. 18 (11) (2006) 1457–1466, doi:10.1109/TKDE.2006.180.
[16] S. Haykin, Neural networks: a comprehensive foundation, Prentice Hall PTR, second ed., Upper Saddle River, NJ, USA, 1998.
[17] 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.
[18] Y. Tian, N. Ali, D. Lo, A.E. Hassan, On the unreliability of bug severity data, Empirical Softw. Eng. 21 (6) (2016) 2298–2323, doi:10.1007/s10664-015-9409-1.
[19] L. Breiman, Random forests, Mach. Learn. 45 (1) (2001) 5–32, doi:10.1023/A:1010933404324.
[20] Z. Zheng, X. Wu, R. Srihari, Feature selection for text categorization on imbalanced data, SIGKDD Explor. Newsl. 6 (1) (2004) 80–89, doi:10.1145/1007730.1007741.
[21] R. Feldman, J. Sanger, Text Mining Handbook: Advanced Approaches in Analyzing Unstructured Data, Cambridge University Press, New York, NY, USA, 2006.
[22] G. Williams, Data Mining with Rattle and R: The Art of Excavating Data for Knowledge Discovery, 2011, doi:10.1007/978-1-4419-9890-3.
[23] N. Japkowicz, M. Shah, Evaluating Learning Algorithms: A Classification Perspective, Cambridge University Press, New York, NY, USA, 2011.
[24] N.K.S. Roy, B. Rossi, Cost-sensitive strategies for data imbalance in bug severity classification: experimental results, in: 2017 43rd Euromicro Conference on Software Engineering and Advanced Applications (SEAA), 2017, pp. 426–429, doi:10.1109/SEAA.2017.71.
[25] C.Z. Yang, C.C. Hou, W.C. Kao, I.X. Chen, An empirical study on improving severity prediction of defect reports using feature selection, in: 2012 19th Asia-Pacific Software Engineering Conference, Vol. 1, 2012, pp. 240–249, doi:10.1109/APSEC.2012.144.
[26] R.K. Saha, J. Lawall, S. Khurshid, D.E. Perry, Are these bugs really "normal"? in: 2015 IEEE/ACM 12th Working Conference on Mining Software Repositories, 2015, pp. 258–268, doi:10.1109/MSR.2015.31.
[27] G. Luo, A review of automatic selection methods for machine learning algorithms and hyperparameter values, Netw. Model. Anal. Health Inf.Bioinf. 5 (1) (2016) 18, doi:10.1007/s13721-016-0125-6.
[28] P. Probst, B. Bischl, A.L. Boulesteix, Tunability: importance of hyperparameters of machine learning algorithms, (2018). arXiv:1802.09596.
[29] K. Facelli, A.C. Lorena, J. Gama, A.C.d. Carvalho, Inteligência Artificial: Uma Abordagem de Aprendizado de Máquina, LTC, Rio de Janeiro, RJ, Brasil, 2015.
[30] F. Wilcoxon, Individual Comparisons by Ranking Methods, Springer New York, New York, NY, 1992, doi:10.1007/978-1-4612-4380-9_16.
[31] N. Lewis, 100 Statistical Tests in R, Easy R Series, Createspace Independent Pub, 2013.
[32] G. Wael, F. Aly, A survey of text similarity approaches, Int. J. Comput. Appl. 68 (13) (2013) 13–18. doi: 10.1.1.403.5446
[33] T. Zhang, J. Chen, G. Yang, B. Lee, X. Luo, Towards more accurate severity prediction and fixer recommendation of software bugs, J. Syst. Softw. 117 (2016) 166–184, doi:10.1016/j.jss.2016.02.034.
[34] S. Robertson, H. Zaragoza, The probabilistic relevance framework: Bm25 and beyond, Found. Trends Inf. Retr. 3 (4) (2009) 333–389, doi:10.1561/1500000019.
[35] G. Yang, T. Zhang, B. Lee, Towards semi-automatic bug triage and severity prediction based on topic model and multi-feature of bug reports, in: 2014 IEEE 38th Annual Computer Software and Applications Conference, 2014, pp. 97–106, doi:10.1109/COMPSAC.2014.16.
[36] A. Heidarian, M.J. Dinneen, A hybrid geometric approach for measuring similarity level among documents and document clustering, 2016, 00, 142–151, doi:10.1109/BigDataService.2016.14
[37] A. Lex, N. Gehlenborg, H. Strobelt, R. Vuillemot, H. Pfister, UpSet: visualization of intersecting sets, IEEE Trans. Visual. Comput.Graph. 20 (12) (2014) 1983–1992, doi:10.1109/TVCG.2014.2346248.
[38] K.K. Chaturvedi, V.B. Sing, P. Singh, Tools in mining software repositories, in: 2013 13th International Conference on Computational Science and Its Applications, 2013, pp. 89–98, doi:10.1109/ICCSA.2013.22.
[39] M. Brhel, H. Meth, A. Maedche, K. Werder, Exploring principles of user-centered agile software development: a literature review, Inf. Softw. Technol. 61 (2015) 163–181, doi:10.1016/j.infsof.2015.01.004.

[40] E.F. de Souza, R. de Almeida Falbo, N.L. Vijaykumar, Knowledge management initiatives in software testing: a mapping study, Inf. Softw. Technol. 57 (2015) 378–391, doi:10.1016/j.infsof.2014.05.016.
[41] K. Chaturvedi, V. Singh, An empirical comparison of machine learning techniques in predicting the bug severity of open and closed source projects, Int. J. Open Source Softw.Processes (IJOSSP) 4 (2) (2012) 32–59, doi:10.4018/jossp.2012040103.
[42] C.Z. Yang, K.Y. Chen, W.C. Kao, C.C. Yang, Improving severity prediction on software bug reports using quality indicators, in: 2014 IEEE 5th International Conference on Software Engineering and Service Science, 2014, pp. 216–219, doi:10.1109/ICSESS.2014.6933548.
[43] M. Sharma, M. Kumari, R.K. Singh, V.B. Singh, Multiattribute based machine learning models for severity prediction in cross project context, in: B. Murgante, S. Misra, A.M.A.C. Rocha, C. Torre, J.G. Rocha, M.I. Falcão, D. Taniar, B.O. Apduhan, O. Gervasi (Eds.), Computational Science and Its Applications – ICCSA 2014, Springer International Publishing, Cham, 2014, pp. 227–241, doi:10.1007/978-3-319-09156-3_17.
[44] N.K.S. Roy, B. Rossi, Towards an improvement of bug severity classification, in: 2014 40th EUROMICRO Conference on Software Engineering and Advanced Applications, 2014, pp. 269–276, doi:10.1109/SEAA.2014.51.
[45] G. Sharma, S. Sharma, S. Gujral, A novel way of assessing software bug severity using dictionary of critical terms, Procedia Comput. Sci. 70 (2015) 632–639. Proceedings of the 4th International Conference on Eco-friendly Computing and Communication Systems
[46] X. Xia, D. Lo, E. Shihab, X. Wang, X. Yang, Elblocker: predicting blocking bugs with ensemble imbalance learning, Inf. Softw. Technol. 61 (2015) 93–106, doi:10.1016/j.infsof.2014.12.006.
[47] S. Gujral, G. Sharma, S. Sharma, Diksha, Classifying bug severity using dictionary based approach, in: 2015 International Conference on Futuristic Trends on Computational Analysis and Knowledge Management (ABLAZE), 2015, pp. 599–602, doi:10.1109/ABLAZE.2015.7154933.
[48] M.N. Pushpalatha, M. Mrunalini, Predicting the severity of bug reports using classification algorithms, in: 2016 International Conference on Circuits, Controls, Communications and Computing (I4C), 2016, pp. 1–4, doi:10.1109/CIMCA.2016.8053276.
[49] A.F. Otoom, D. Al-Shdaifat, M. Hammad, E.E. Abdallah, Severity prediction of software bugs, in: 2016 7th International Conference on Information and Communication Systems (ICICS), 2016, pp. 92–95, doi:10.1109/IACS.2016.7476092.
[50] K.K. Sabor, M. Hamdaqa, A. Hamou-Lhadj, Automatic prediction of the severity of bugs using stack traces, in: Proceedings of the 26th Annual International Conference on Computer Science and Software Engineering, CASCON '16, IBM Corp., Riverton, NJ, USA, 2016, pp. 96–105.
[51] K. Jin, A. Dashbalbar, G. Yang, J.W. Lee, B. Lee, Bug severity prediction by classifying normal bugs with text and meta-field information, Adv. Sci. Technol. Lett. 129 (2016) 19–24, doi:10.14257/astl.2016.129.05.
[52] T. Choeikiwong, P. Vateekul, Improve accuracy of defect severity categorization using semi-supervised approach on imbalanced data sets, in: Proceedings of the International MultiConference of Engineers and Computer Scientists, Vol. 1, 2016.
[53] K. Jin, A. Dashbalbar, G. Yang, B. Lee, Improving predictions about bug severity by utilizing bugs classified as normal, Contemp. Eng. Sci. 9 (2016) 933–942, doi:10.12988/ces.2016.6695.
[54] K. Jin, E.C. Lee, A. Dashbalbar, J. Lee, B. Lee, Utilizing feature based classification and textual information of bug reports for severity prediction, Information 19 (2) (2016) 651–659.
[55] V.B. Singh, S. Misra, M. Sharma, Bug severity assessment in cross project context and identifying training candidates, J. Inf. Knowl. Manage. 16 (01) (2017) 1750005, doi:10.1142/S0219649217500058.
[56] R. Pressman, Software Engineering: A Practitioner's Approach, seventh ed., McGraw-Hill, Inc., New York, NY, USA, 2010.
[57] I. Guyon, A. Elisseeff, An introduction to variable and feature selection, J. Mach. Learn. Res. 3 (2003) 1157–1182.
[58] T. Menzies, A. Marcus, Automated severity assessment of software defect reports, in: 2008 IEEE International Conference on Software Maintenance, 2008, pp. 346–355, doi:10.1109/ICSM.2008.4658083.
[59] N.V. Chawla, K.W. Bowyer, L.O. Hall, W.P. Kegelmeyer, Smote: synthetic minority over-sampling technique, J. Artif. Int. Res. 16 (1) (2002) 321–357.
[60] L.A. Jeni, J.F. Cohn, F.D.L. Torre, Facing imbalanced data–recommendations for the use of performance metrics, in: 2013 Humaine Association Conference on Affective Computing and Intelligent Interaction, 2013, pp. 245–251, doi:10.1109/ACII.2013.47.
[61] A. Ben-David, Comparison of classification accuracy using Cohen's Weighted Kappa, Expert Syst. Appl. 34 (2) (2008) 825–832, doi:10.1016/j.eswa.2006.10.022.
[62] K. Krippendorff, Computing Krippendorff's Alpha-Reliability, Departamental Papers (ASC), 2011.
[63] A. Fernandez, S. Garcia, F. Herrera, N.V. Chawla, Smote for learning from imbalanced data: progress and challenges, marking the 15-year anniversary, J. Artif. Intell. Res. 61 (2018) 863–905.
[64] B. Kitchenham, L. Madeyski, D. Budgen, J. Keung, P. Brereton, S. Charters, S. Gibbs, A. Pohthong, Robust statistical methods for empirical software engineering, Empir. Softw. Eng. 22 (2) (2017) 579–630, doi:10.1007/s10664-016-9437-5.
[65] A. Nigam, B. Nigam, C. Bhaisare, N. Arya, Classifying the bugs using multi-class semi supervised support vector machine, in: International Conference on Pattern Recognition, Informatics and Medical Engineering (PRIME-2012), 2012, pp. 393–397, doi:10.1109/ICPRIME.2012.6208378.
[66] G. Wang, C. Li, W. Wang, Y. Zhang, D. Shen, X. Zhang, R. Henao, L. Carin, Joint embedding of words and labels for text classification, 2018. arXiv:1805.04174
[67] I. Goodfellow, Y. Bengio, A. Courville, Deep Learning, MIT Press, 2016.
[68] Z. Chen, B. Liu, Lifelong machine learning, Synth. Lect. Artif. Intell.Mach. Learn. 10 (3) (2016) 1–145.
[69] R.K. Saha, S. Khurshid, D.E. Perry, Understanding the triaging and fixing processes of long lived bugs, Inf. Softw. Technol. 65 (2015) 114–128, doi:10.1016/j.infsof.2015.03.002.
[70] S. Thiago, R. Leonardo, G.M. André, A.J. M., M. Fernando, M. Wagner, V. Felipe, A quantitative analysis of the temporal effects on automatic text classification, J. Assoc. Inf. Sci. Technol. 67 (7) (2016) 1639–1667, doi:10.1002/asi.23452.
[71] F. Al-Hawari, H. Barham, A machine learning based help desk system for IT service management, J. King Saud Univ., 2019. http://www.sciencedirect.com/science/article/pii/S1319157819300515.
[72] B.A. Kitchenham, D. Budgen, O.P. Brereton, Using mapping studies as the basis for further research - a participant-observer case study, Inf. Softw. Technol. 53 (6) (2011) 638–651, doi:10.1016/j.infsof.2010.12.011.
[73] J. Demšar, Statistical comparisons of classifiers over multiple data sets, J. Mach. Learn. Res. 7 (2006) 1–30.
[74] N. Christianini, J. Shawe-Taylor, An Introduction to Support Vector Machines and Other Kernel-Based Learning Methods, Vol. 18, Cambridge University Press, Cambridge, 2000.
[75] K.P. Murphy, Machine Learning: A Probabilistic Perspective, The MIT Press, 2012.
[76] S.J. Russell, P. Norvig, Artificial intelligence: a modern approach, 2010, arXiv:1011.1669v3.
[77] K.S. Beyer, J. Goldstein, R. Ramakrishnan, U. Shaft, When is "nearest neighbor" meaningful? in: Proceedings of the 7th International Conference on Database Theory, ICDT '99, Springer-Verlag, London, UK, UK, 1999, pp. 217–235.
[78] A.M. Kibriya, E. Frank, B. Pfahringer, G. Holmes, Multinomial naive Bayes for text categorization revisited, in: X. Webb, G.I.a. Yu (Eds.), AI 2004: Advances in Artificial Intelligence, Springer Berlin Heidelberg, Berlin, Heidelberg, 2005, pp. 488–499.
[79] N.V. Chawla, Data mining for imbalanced datasets: an overview, in: Data Mining and Knowledge Discovery Handbook, Springer US, Boston, MA, 2009, pp. 875–886, doi:10.1007/978-0-387-09823-4_45.
[80] Y. Yang, J.O. Pedersen, A comparative study on feature selection in text categorization, in: Proceedings of the Fourteenth International Conference on Machine Learning, ICML '97, Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1997, pp. 412–420.