Learning to Predict Long-Lived Bug Report Severity Level on Free/Libre Open Source Software

Qualification Exam

Ph.D. Student: Luiz Albero Ferreira Gomes

Principal Advisor: Prof. Mario Lúcio Côrtes

Co-advisor: Prof. Ricardo da Silva Torres

Institute of Computing

UNICAMP

September 3, 2018

Abstract

Software evolution and maintenance activities in today Free/Libre Open Source Software rely primarily on information extracted from reports registered in bug tracking systems. The severity level attribute of a bug report is considered one of the most critical variables for planning these activities. For example, this attribute 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. An incorrect assignment of severity level may generate a significant impact on the maintenance process. Nevertheless, methods to assign severity level remains essentially manual with a high degree of subjectivity. Because it depends on the experience and expertise of whom have reported or reviewed the bug report, it may be considered a quite error-prone process. Due to its evident importance, both business and academic community have demonstrated a great interest in this topic, which has been associated with an extensive investigation towards the proposal of methods to predict the bug report severity automatically. We show in a systematic mapping review annexed to this qualification exam document that traditional machine learning methods and text mining techniques have been playing a successful role in these efforts. This review also reveals that two important challenges in this area are imbalanced data and high dimensionality in datasets used for prediction. Besides, the proposed approaches have not been considered the lifecycle duration and the temporal context of bug reports. Studies show that short-lived bug reports represent around 50% of bug reports in some relevant FLOSS projects, and they are closed within seven days for many reasons (e.g., fixed, duplicated, or invalid). Thus, efforts to predict severity level of short-lived bug reports seems meaningless and useless. Conversely, the number of long-lived bugs is high and over 90% of them adversely affect the user's experience. The same studies suggest that many long-lived bugs could be fixed quickly through careful triaging and prioritization, if developers could, for example, predict their severity, in advance. This research work aims to bridge this gap by proposing learning models for long-lived bug report severity prediction that takes advantage of temporal context, an issue which is of paramount importance for this kind of bug. We propose two research venues to build learning models: (i) develop new feature selection methods or extend existing ones which take into account the temporal context of a long-lived bug report; and (ii) develop new data-driven approaches for long-live bug severity prediction which consider the temporal context of a long-lived bug report. Furthermore, both proposed approaches must be able to address imbalanced data and high dimensionality in bug reports datasets used for severity prediction.

I. INTRODUCTION

Bug Report Tracking Systems (BTS) have a major role in the evolution and maintenance process in many software development settings, both in Closed Source Software (CSS) and Free/Libre Open Source Software (FLOSS) scenarios. This is especially true in FLOSS, which is characterized by the existence of many of users and developers with different levels of expertise spread out around the world, who might create or be responsible for several bug reports [1].

A user interacts with a BTS often through a simple mechanism called bug report form. This form enables him to request changes, to report bugs or to ask for support in software product [2]. Initially, he or she should inform a short description, a long description, a type (e.g., bug, new feature, enhancement, and

task) and an associated severity level (e.g., blocker, critical, major, minor, and trivial). Later, a development team member reviews this report and, if it is not refused for some reason (e.g., request duplication), he or she provides additional information in the bug report form, indicating, for example, its priority and the person who will be responsible for the bug report.

Typically, the number of bug reports in large and medium software FLOSS projects is frequently very large [3]. Therefore, efforts have been made to implement intelligent software assistants to help developers and maintenance personnel in defining more accurately the attribute values in a bug report form [4].

Severity level shifts throughout bug report *lifecycle* may have an adverse effect on the planning of maintenance activities. For example, the maintenance team could be demanded to address less significant bug reports before most important ones. Thus, severity level information is recognized as a critical variable in the equation to estimate a prioritization of bug reports [2], as it defines how soon the bug reports need to be addressed [5]. However, the severity level assignment remains mostly a manual process, which is mainly based on the experience and expertise of the person who has opened the bug report [1], [2], [5]. Consequently, it is a time-consuming and error-prone process with a high degree of subjectivity.

In a comprehensive systematic mapping review of recent research efforts on automatically bug report severity prediction, we pointed out that these efforts have overlooked bug reports lifecycle duration (Annex A). In another study, Saha et al. [6] examined seven FLOSS projects and noted that around 50% of bug reports are closed within seven days for many reasons (e.g., fixed, duplicated, or invalid). Thus, efforts to predict severity level of these short-lived bug reports seems to be useless. The same study also reveals that there are a considerable number of long-lived bugs in each system and over 90% of them adversely affect the user's experience. Besides, the study suggests which many long-lived bugs could be fixed quickly through careful triaging and prioritization, if developers could, for example, predict their severity, in advance.

The definition of long-lived is subjective since the time threshold for deciding whether a bug is long-lived or short-lived could vary across projects, practioners or studies. Saha et al. [6], for instance, consider that a long-lived bug should survive over at least two release cycles of a product which corresponds at a minimum one year in the FLOSS investigated in this paper. In this interval, it is very likely that the long-lived bug report information will evolve making them more realistic. Because of this, considering this temporal context is essential for severity predicting of a long-lived bug report. For example, Figure 1, the temporal context evolution do bug report HADOOP-8855. Throughout its lifecycle (from t_0 to t_{n+1}), the user or development team member might update its attribute values (e.g., he or she writes a new comment or changes the description). Furthermore, others bug reports could reference to bug report HADOOP-8855, so creating many dependency relations among them. Because of these temporal context changes, the severity of the bug report could change from trivial to critical.

Furthermore, our mapping review reveals the current research efforts have strongly relied on traditional techniques in Machine Learning (ML), Feature Selection (FS) and Text Mining (TM). Yielded results may be improved, for example, by applying state-of-the-art techniques, based, for example, on data driven approaches. Our review also reveals that two important research challenges in this area refer to imbalanced and high dimensional data, commonly available in datasets used for prediction.

This research work aims to bridge this gap by proposing learning models for long-lived bug report severity prediction that takes advantage of temporal context, an issue which is of paramount importance for this kind of bug. We propose two research venues to build learning models: (i) develop new feature selection methods or extend existing ones which take into account the temporal context of a long-lived bug report; and (ii) develop new data-driven approaches for long-live bug severity prediction which consider the temporal context of a long-lived bug report. Furthermore, both proposed approaches must be able to address imbalanced data and high dimensionality in bug reports datasets used for severity prediction.

The proposal is organized as follows. Section II describes the research problem and hypothesis. Section III defines the goals of our work. Section IV presents the background concepts. Section V discusses related works that are relevant to our research. Section VI details our thesis proposal. Section VII describes our
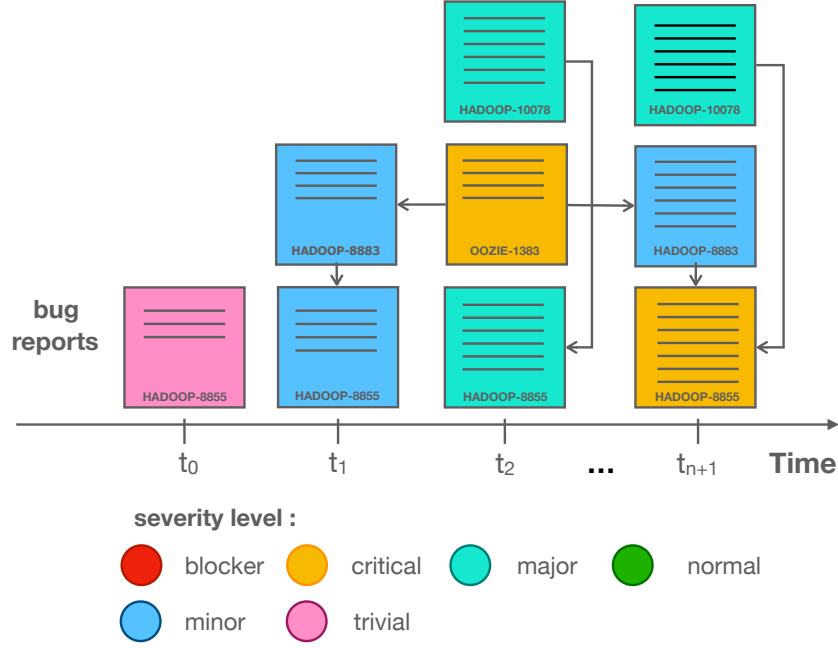
Fig. 1: Bug report temporal evolution.

research method. Section VIII presents the scheduled time-line. Finally, Annex A presents our systematic mapping review of bug report severity prediction in FLOSS projects.

II. PROBLEM AND HYPOTHESIS

Bug report severity prediction is typically modeled as a classification problem. After the preprocessing step, bug reports attributes will be converted into a feature vector. Formally, in a training dataset with a set of N bug reports or documents, each document is represented as an *m*-dimensional vector $x = (\{x_1, x_2, ..., x_m\})$, where $x_j \in X_j$ corresponds to the value of random variable $X_j (j = 1, 2, \ldots, m)$. The set $X = X_1 \times X_2 \times \ldots \times X_m$ denotes the *input space*. This set contains all possible inputs to the learning algorithm. Classification tasks require that the machine learning method produces an output. The *output space* is defined in a similar way, where $y \in Y$ denotes the value of the output random variable Y. The output space can also have more than one dimension and expressing this is analogous to the input space. Variables X_j are often referred to as *features* or *dimensions*, **x** as *sample* or *instance*, and Y as *label* or *class*.

Considering the temporal contexts of long-lived bug reports both, the *m*-dimensional vector x and the output random variable y may evolve in time. To represent correctly such temporal context, x will be defined as $x_t = \{\{x_1^{t_0}, x_2^{t_0}, ...x_m^{t_0}\}, \{x_1^{t_1}, x_2^{t_1}, ...x_m^{t_1}\}, ..., \{x_1^{t_n}, x_2^{t_n}, ...x_m^{t_n}\}\}\}$ and y will be defined as $y_t = \{y^{t_0}, y^{t_1}, ..., y^{t_n}\}$ where $n = (0, 1, 2, 3, ...)$ and $T = \{t_0, t_1, t_2, ..., t_n\}$ be set timestamps when both test and training documents were create or updated. Besides, the set of classes $y^t \in (c_1, c_2, ..., c_i)$, where $(i = 2, 3, ..., k)$ depends on the BTS in which FLOSS is hosted. Typically k ranges between five and six classes (e.g., blocker, critical, major, normal, minor, trivial).

In the real world, bug reports training and test datasets are imbalaced [7] and high-dimensional [8]. The input space (x_t) in the temporal context of long-lived bug report makes the high-dimensional problem more difficult since now we could have to consider n sets of x.

The problem described in before paragraphs will be investigated considering the following research hypotheses:

H1. The temporal context of long-lived bug report leads to the performance improvement of classification systems used to bug report severity prediction.

H1. The feature selection considering the temporal context of long-lived bug reports mitigate the effects of imbalanced data and high-dimensional in classification systems used to severity prediction. H3. The temporal context modeling of long-lived bug reports using data-driven methods leads to the performance improvement of classification systems used to bug report severity prediction.

III. GOALS AND CONTRIBUTIONS

The general goal of this research project is to build learning models that allow predicting the severity of long-lived bug reports. Those models are expected to consider the temporal context of the bug report and address two common problems in bug reports dataset: imbalanced and high-dimensional data. In summary, the following objectives are envisioned:

G_1 Study of the state of the art in bug report severity prediction in FLOSS projects, problems, and opportunities (completed).

$G_{1.1}$ Write a systematic mapping review on bug report severity prediction in FLOSS projects (Annex A).

$G_{1.2}$ Identify the main problems on bug report severity prediction (Annex A, pages 22 and 23).

G_2 Select one or more problems and advance the state of the art in this area (partially completed).

$G_{2.1}$ The following problems were selected:

    a. Prediction severity considering the temporal context of long-lived bug reports.
    b. Imbalanced temporal context datasets in long-lived bug report severity prediction.
    c. High dimensionality in temporal context bug reports datasets in long-lived bug report severity prediction.

$G_{2.2}$ Develop new feature selection methods, which address the temporal context issues of a long-lived bug report, imbalanced data and high dimensionality in bug reports datasets used to severity prediction.

$G_{2.3}$ Develop new data-driven approaches for long-live bug severity prediction which take into account the temporal context of a long-lived bug report, imbalanced data and high dimensionality in bug reports datasets.

The contributions of this research project for computer science are:

1) New learning models that will use the temporal evolution information of bug reports for long-lived bug report severity prediction which so far has not been considered in any proposed solution for this problem.

2) New learning models based on novel feature selection methods which address the temporal context of a long-lived bug report, imbalanced data, and high-dimensionality in bug reports datasets used for bug report prediction.

3) New learning models based on novel data-driven approaches to long-lived bug report prediction which address the temporal context of a bug report, imbalanced data, and high-dimensionality in bug reports datasets.

The contributions of this research project for the FLOSS maintenance area are:

1) New learning models which effectively address long-lived bug report and improve the maintenance process of FLOSS projects. This is important because long-lived bugs may propagate through more than one major releases of a product, affecting negatively the user experience negatively with many major versions [6].

IV. BACKGROUND

Our systematic review (Anenx A, section II) provide a explanation of a rich set of concepts and terminology related to bug report severity prediction area. This section briefly introduce some specific concepts, which were not addressed in this review.

A. Deep Learning

The proposed approaches in the literature have strongly adopted traditional ML algorithms for bug report severity prediction (Annex A, Table XIII, page 18). However, these algorithms need careful feature engineering and substantial domain expertise to design a feature extractor that transforme raw data into a feature vectors from which a classifier could predict a class to an input instance [9]. On the other hand, deep learning has been proved to be very good at discovering complex structures in high-dimensional [9], because this learning method enables a machine to read raw data and to discover its representations automatically.

Deng and Yu [10] define deep learning as "a sub-field within machine learning that is based on algorithms for learning multiple levels of representation to model complex relationships among data". The Convolutional Neural Network (CNN) and the Recurrent Neural Network (RNN) are two mainstream architecture for sentence and document modelling [9]. A CNN can be defined as a feedforward neural network, which consists of an input and an output layer, as well as multiple hidden layers. It is capable of capturing the global and the local features and significantly enhancing the efficiency and accuracy [11]. On the other hand, a RNN is suitable for modeling sequential data because connections between nodes form a directed graph along a sequence. There are loops and memories in RNN to remember former computations. Long Short-Term Memory (LSTM) and Gated Recurrent Unit (GRU) are two examples of RNN [11].

B. Word Embedding

Bag-of-Words was the vector extraction model most used for bug severity prediction in the literature (Annex A, Table XII, page 18). This model was based on counting frequencies of occurrences of shot symbols sequences of length up to N (N-grams). The number of possible N-grams is on the order of V^N, where V is the vocabulary size, so taking into account a context of more than a handful of words would require very large training corpora [9].

A principal component of the neural-network approach is the use of embeddings - representing each feature as a vector in a low dimensional space. Word embeddings are a class of techniques which represents individual words as real-valued vectors in a predefined vector space. It learns a vector representation of words by mapping semantic information into a geometric word embedding space. Such models, the vector representation *w* of a given the word is usually learned through a fixed context window *W*. Each word is mapped to one vector, and the vector values are learned in a way that resembles a neural network. There is two-word embeddings representation:

- **Word2Vec [12]** is a statistical method for learning a standalone word embedding from text corpus which involves analysis of learned vectors and the exploration of vector math on the representation of words.
- **GloVe [13]** is a statistical method which captures semantic information from the fixed context window and global corpus statistics through the word co-occurrence probabilities.

C. Genetic programming algorithm

Few approaches for bug severity prediction used some method for feature selection. Common methods used are include in Table XI (Annex A, page 17). Recent and promising methods for feature selection based on Genetic Programming (GP) have been successfully used in Automatic Document Classifying (ADC) [14]–[16]. This technique provides a framework for automatically creating a working computer program from a high-level problem statement of the problem. GP achieves this goal of automatic programming by genetically breeding a population of computer programs using the principles of Darwinian natural selection and biologically inspired operations [17]. As a population-based evolutionary computation technique, GP typically follows these steps [14]:

1) Generate an initial population of individuals composing of primitive functions and terminals of the problem.

2) Iteratively performs the following sub-steps until a stopping criterion is met:

    a) Evaluation: each individual is executed, and its fitness is calculated based on a predefined fitness function.

    b) Selection: select one or two individuals from the population with a probability based on fitness to participate in the evolution step.

    c) Evolution: create new individuals for the new population by applying the following genetic operators with specific probabilities: (i) copy the selected individuals to the new population (reproduction); (ii) create new offsprings by recombining randomly chosen parts from two selected programs (crossover); and (iii) create a new offspring program by randomly mutating a randomly chosen part of one selected program (mutation).

3) Return the program with the highest fitness as the best solution.

Applying GP to find a solution of a problem demands the definition of a terminal set, a function set, a fitness function, a stopping criterion and control parameters such as population size, crossover and mutation probabilities.

V. Related Work

This section presents includes more related works to those presented in our mapping review. Works presented here has large potential to be adapted to our proposed learning model.

A. Graph-based text representation

The approaches presented in this section typically model a text document as a graph, in which a vertex presents a term from the text and an edge represents a relationship between terms.

Dourado [18] proposed a Bag Textual Graph (BoTG), an accurate graph-based text representation model that combines term counting with also locality principles such as proximity and the order terms within a text, in an efficient general-purpose vector space representation. The objective is to model textual documents as graphs and to project them in a vector space that has lower dimensionality and higher efficiency than the Bag-of-Words model, with comparable accuracy to the graph models but with more general applicability. Authors performed experiments using four textual datasets (Reuters-21578, 4univerties, 20-newsgroup, and J-series) and three classifiers (kNN, Random Forest, and SVM) and demonstrate that the proposed method is effective for different scenarios. This model, however, does not map documents relationships which are essential to de our problem.

Defferrard et al. [19] present a formulation of CNN in the context of spectral graph theory, which provides the necessary mathematical background and efficient numerical schemes to design fast localized convolutional filters on graphs. Importantly, the proposed technique offers the same linear computational complexity and constant learning complexity as classical CNNs, while being universal to any graph structure. Although, this deep learning system technique is limited to learn local, stationary, and compositional features on graphs.

Kipf et al. [20] present a scalable approach for semi-supervised learning on graph-structured data that is based on an efficient variant of convolutional neural networks which operate directly on graphs. According to authors, their model scales linearly in number of graph edges and learns hidden layer representations that encode both local graph structure and feature nodes. Unlike our learning models, Defferrard et al. [19] and Kipf et al. [20] does not take into account the temporal context of documents which is mapped as graph node.

B. Temporal context aware methods

Salles et al. [21] provided evidence of the existence of temporal effects in three textual datasets, reflected by variations observed over time in class distribution, in pairwise class similarities, and in the relationships between terms and classes. They also discuss that these temporal effects affect each analyzed

dataset differently and that they restrict the performance of each considered text classification algorithm. This study motivate us to investigate that the temporal evolution of bug reports and bug repositories information also may affect the performance of classifiers for bug report severity prediction.

In another study, Salles et al. [22] proposed a machine learning methodology to learn the *temporal weighting function* (TWF) automatically and they also propose new strategies to incorporate the TWF into ADC algorithms, which we call *temporally-aware* classifiers. Experiments showed that fully-automated temporaly-aware classifiers achieved significant gains (up to 17%) when compared to their non-temporal counterparts, even outperforming some state-of-the-art algorithms (e.g., SVM) in mosts cases, with large reductions in execution time. While authors worked on static pure textual datasets, our research project (ACM-DL, MEDLINE, and AG-NEWS) will address temporal context issues on dynamic bug reports repositories which contain qualitative, quantitative and unstructured texts features.

C. Feature Selection Methods

Tran et al. [14] tested and compared GP for feature construction and selection on high-dimensional classification problems over seven high-dimensional gene datasets (e.g., Leukemia and Prostate). The results showed that GP could reduce the dimensionality and maintain or even increase the classification accuracy in most cases. In another study, Viegas et al. [16] proposed a novel feature selection strategy based on GP. Their approach combines the most discriminative features sets selected by distinct feature selection metrics to obtain a more effective and impartial set of the most discriminative features. According to the authors, their approach works well with balanced and unbalanced data. Our research proposal differs from [14], [16] by considering temporal context information as discriminative factor for feature selection

D. Data Driven Methods

Zhang et al. [23] and Yang et al. [24] used CNN for text classification. Zhang et al. [23] obtained competitive results applying character-level CNN in several large-scale datasets (e.g., Yahoo! Answers). In this turn, Yang et al. [24] combined word-embeddings for feature extraction and CNN and outperformed SVM and TF-IDF in Twitter election classification. Lai et al. [25], on the other hand, worked with another deep learning network. They applied RNN to capture contextual information and max-pooling to decide words play critical roles in text classification. Their approach outperformed state-of-the-art models (e.g., CNN) on four textual datasets (20News, Fudan, ACL, and SST). Unlike our learning models, [23]–[25] approaches also did not take into account the temporal context and relationships among documents in datasets

VI. Proposal

As previously state, we aim to conceive learning models for long-lived bug severity prediction with two main requirements to attain our objectives: (i) they must be able to handle data that changes over time; (ii) they must be resilient to high-dimensionality in datasets used for prediction; (iii) they must be resilient do imbalanced data in datasets used for prediction.

Figure 2 presents the main modules of our proposed learning model to meet de requirements outlines above. Module 1 will be in charge of downloading bug reports from FLOSS repositories (e.g., based on Bugzilla, Github, and Jira) and store them in a raw data format into a disk. Module 2 will preprocess these bug reports, by type checking, normalizing, performing fix and imputation activities [26], and transforming them in a consistent data format suitable for machine learning classifiers.

In order to organize bug reports data, as well as, its relations, we will investigate methods that can learn from dynamic graphs. As a starting point, we propose to extend the BoTG [18] to represent the temporal context of long-lived bug reports. An advantage of this model for our problem is to organize data in low-dimensionality. This model also seems to be promising to represent dependencies among documents.

Next, we will investigate feature selection methods which are suitable for handling imbalance and high dimensional data. Initially, we will extend feature selection methods based on GP [14]–[16] to consider the
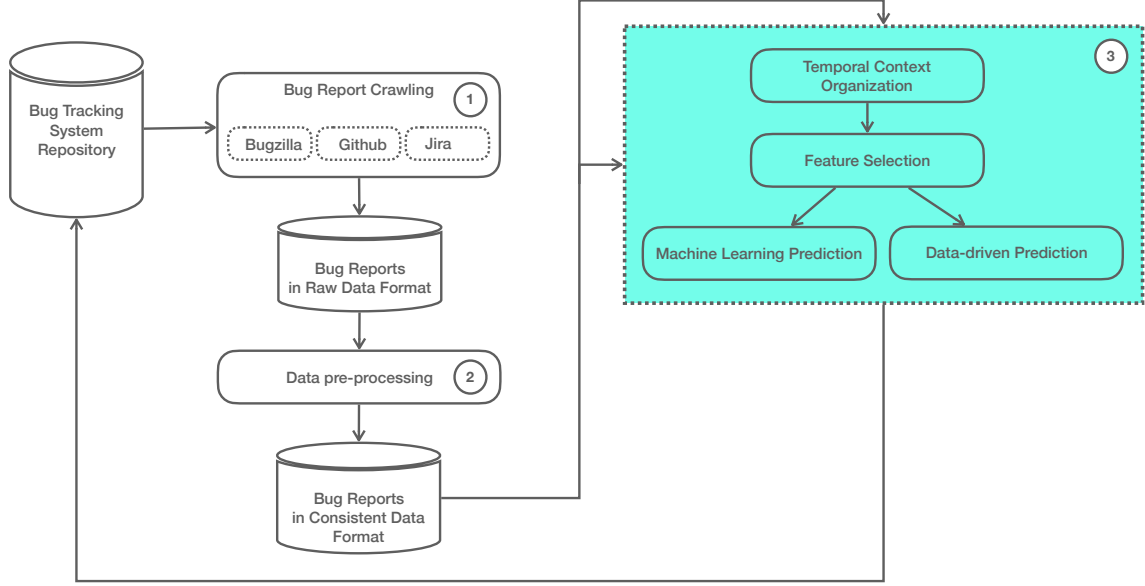
Fig. 2: Proposed learning model.

temporal context of long-lived bug reports. Such methods have been well-succeed to address imbalanced and high-dimensionality in textual datasets for document classification.

To accomplish the prediction task of our learning model (module 3), we will investigate two approaches (i) traditional machine learning methods; and (ii) data-driven methods. Proposed approaches reviewed in our systematic mapping have commonly used traditional machine learning methods for bug report severity prediction. Initially, we use these traditional methods in our learning model. After this activity, we will be able to compare our yielded results with results published in the literature. Also, we plan to investigate and use data-driven methods in the prediction task. Well-succeed approaches for text classifying using data-driven methods, such deep learning, motivate us to follow this research avenue: ([24], [27], [28], [23], and [25]).

VII. RESEARCH METHOD

The primary research method of this research project is based on empirical experimentation. As in typical methodologies used in ML experiments, we follow the next steps:

- **Data Extraction**: this step in the experimental research encompasses selecting FLOSS projects to serve as the data source, studying and interpreting its data structure, and finally extracting relevant data from its repository (feature extraction).
- **Data preprocessing**: Raw data previously collected from selected FLOSS repositories is not correctly structured to serve as input to ML algorithms, they were not in a consistent data format. The classical way to address this problem is to run preprocessing procedures to extract, organize and structure relevant features out of the raw data.
- **Training and testing**: Training and testing steps start with partitioning the already preprocessed dataset in two disjoint subsets: a subset from training, typically with 80% of the bug reports, and a subset for testing, with the remaining 20% of the bug reports.
- **Evaluation**: this step in the experimental research encompasses the evaluation of experiments results and validation utilising statistical tests (Annex A, page 6).

A. Experimental Setup

Datasets: We will train and test our proposed model on datasets from FLOSS projects outlined in Table VII in our systematic mapping review (Annex A). These relevant FLOSS projects are hosted in

distinct BTS (Bugzilla, Google, and Buzilla) and their bug reports are free available providing datasets with different properties to validate our learning model.

ML algorithms: We will build our proposed model using traditional ML algorithms outlined in Table XIII in our systematic mapping review (Annex A). As starting point, we will investigate four widely used algorithms for bug report severity prediction, according to our review: KNN, Näive Bayes, Näive Bayes Multinomial, and SVM.

Evaluation metrics: We will evaluate our proposed model using metrics outlined in Table XIV in our systematic mapping review (Annex A). As starting point, we will evaluate our learning model performance using the three most used metrics: precision, recall e f-measure.

Data driven algorithms: As starting point, will use two data-driven algorithms based on deep learning family considered mainstream in document modeling and well-succeed applied in text classification [9]: CNN and RNN.

Baselines: We compare our learning model to the approaches proposed in papers outlined in Table IV in our systematic mapping review (Annex A).

VIII. PROJECT TIMELINE

Table I describes the project timeline that includes the doctoral degree requirements and research activities.

TABLE I: Project time line

Activity	2016	2017		2018		2019		2020
	2	1	2	1	2	1	2	1
(a) Literature review	■	■	■	■	■	■	■	■
(b) Study of ML and TM Methods	■							
(c) Implementation of crawlers of bug reports	■							
(d) Development of scripts to run and analyze ML and TM experiments		■	■					
(e) Write a Systematic Mapping Review			■	■				
(f) Qualification Examination					■			
(g) Study of Genetic Programming and Deep Learning					■			
(h) Model temporal context of long-lived bug report						■	■	
(i) Design, implement execute and evaluate experiments						■	■	
(j) Publish papers						■		
(l) Defense of Thesis								■

The research steps are commented next:

b. Study of main ML and TM methods, as well as methodologies to effective using these methods (completed).

c. Implementation of Java Program to bug reports extracting from Bugzilla and Jira repositories, and converting them to a consistent format (completed).

d. Designing and implementation of experiments to learn and to test traditional ML algorithms to predict bug severity from bug reports collected from Bugzilla and Jira repositories. These experiments evolved implementation of the complete machine learning pipeline: data collecting, data pre-processing, training, and testing (completed).

e. Write a systematic mapping review of bug report severity prediction (Annex A) which yielded a paper that will be submitted to a journal (completed).

g. Study of main GP and Deep Learning methods which can be applied to our problem.

h. Model temporal context of long-lived bug report using.

i. Design, implement, execute and evaluate experiments to validate our proposed our learning model.

j. Publish the results of our research in selective journals and conferences.

REFERENCES

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

[2] Y. Tian, D. Lo, and C. Sun, "Information Retrieval Based Nearest Neighbor Classification for Fine-Grained Bug Severity Prediction," in *2012 19th Working Conference on Reverse Engineering*, oct 2012, pp. 215–224.

[3] A. Lamkanfi, S. Demeyer, Q. D. Soetens, and T. Verdonckz, "Comparing mining algorithms for predicting the severity of a reported bug," *Proceedings of the European Conference on Software Maintenance and Reengineering, CSMR*, pp. 249–258, 2011.

[4] Y. Tian, D. Lo, X. Xia, and C. Sun, "Automated prediction of bug report priority using multi-factor analysis," *Empirical Software Engineering*, vol. 20, no. 5, pp. 1354–1383, oct 2015.

[5] A. Lamkanfi, S. Demeyer, E. Giger, and B. Goethals, "Predicting the severity of a reported bug," *Proceedings - International Conference on Software Engineering*, pp. 1–10, 2010.

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

[7] N. K. S. Roy and 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)*, Aug 2017, pp. 426–429.

[8] C. Z. Yang, C. C. Hou, W. C. Kao, and 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, Dec 2012, pp. 240–249.

[9] Y. LeCun, Y. Bengio, and G. Hinton, "Deep learning," *Nature*, vol. 521, pp. 436 EP –, May 2015. [Online]. Available: http://dx.doi.org/10.1038/nature14539

[10] L. Deng and D. Yu, *Deep Learning: Methods and Applications*. Hanover, MA, USA: Now Publishers Inc., 2014.

[11] A. Karatzoglou and B. Hidasi, "Deep learning for recommender systems," in *Proceedings of the Eleventh ACM Conference on Recommender Systems*, ser. RecSys '17. New York, NY, USA: ACM, 2017, pp. 396–397.

[12] Y. Goldberg and G. Hirst, *Neural Network Methods in Natural Language Processing*. Morgan & Claypool Publishers, 2017.

[13] J. Pennington, R. Socher, and C. D. Manning, "Glove: Global vectors for word representation," in *In EMNLP*, 2014.

[14] B. Tran, B. Xue, and M. Zhang, "Genetic programming for feature construction and selection in classification on high-dimensional data," *Memetic Computing*, vol. 8, no. 1, pp. 3–15, Mar 2016.

[15] B. Xue, M. Zhang, W. N. Browne, and X. Yao, "A survey on evolutionary computation approaches to feature selection," *IEEE Transactions on Evolutionary Computation*, vol. 20, no. 4, pp. 606–626, Aug 2016.

[16] F. Viegas, L. Rocha, M. Gonçalves, F. Mourão, G. Sá, T. Salles, G. Andrade, and I. Sandin, "A genetic programming approach for feature selection in highly dimensional skewed data," *Neurocomputing*, vol. 273, pp. 554 – 569, 2018.

[17] N. Nedjah, A. Abraham, and L. de Macedo Mourelle, *Genetic Systems Programming: Theory and Experiences*, 1st ed. Springer Publishing Company, Incorporated, 2009.

[18] Ícaro Calvancate Dourado, "Bag of Textual Graphs," Master's thesis, University of Campinas, Campinas, Brazil, 2016.

[19] M. Defferrard, X. Bresson, and P. Vandergheynst, "Convolutional neural networks on graphs with fast localized spectral filtering," in *Proceedings of the 30th International Conference on Neural Information Processing Systems*, ser. NIPS'16. USA: Curran Associates Inc., 2016, pp. 3844–3852. [Online]. Available: http://dl.acm.org/citation.cfm?id=3157382.3157527

[20] T. N. Kipf and M. Welling, "Semi-supervised classification with graph convolutional networks," *CoRR*, vol. abs/1609.02907, 2016. [Online]. Available: http://arxiv.org/abs/1609.02907

[21] S. Thiago, R. Leonardo, G. M. André, A. J. M., M. Fernando, M. Wagner, and V. Felipe, "A quantitative analysis of the temporal effects on automatic text classification," *Journal of the Association for Information Science and Technology*, vol. 67, no. 7, pp. 1639–1667.

[22] T. Salles, L. Rocha, F. Mourão, M. Gonçalves, F. Viegas, and W. Meira, "A two-stage machine learning approach for temporally-robust text classification," *Information Systems*, vol. 69, pp. 40 – 58, 2017.

[23] X. Zhang, J. J. Zhao, and Y. LeCun, "Character-level convolutional networks for text classification," *CoRR*, vol. abs/1509.01626, 2015.

[24] X. Yang, C. MacDonald, and I. Ounis, "Using word embeddings in twitter election classification," *CoRR*, vol. abs/1606.07006, 2016. [Online]. Available: http://arxiv.org/abs/1606.07006

[25] S. Lai, L. Xu, K. Liu, and J. Zhao, "Recurrent convolutional neural networks for text classification," in *Proceedings of the Twenty-Ninth AAAI Conference on Artificial Intelligence*, ser. AAAI'15. AAAI Press, 2015, pp. 2267–2273.

[26] E. de Jonge and M. van der Loo, "An introduction to data cleaning with R," *Statistics Netherlands*, p. 53, 2013. [Online]. Available: http://cran.r-project.org/doc/contrib/de_Jonge+van_der_Loo-Introduction_to_data_cleaning_with_R.pdf

[27] M. Jiang, Y. Liang, X. Feng, X. Fan, Z. Pei, Y. Xue, and R. Guan, "Text classification based on deep belief network and softmax regression," *Neural Computing and Applications*, vol. 29, no. 1, pp. 61–70, Jan 2018.

[28] D. S. Sachan, M. Zaheer, and R. Salakhutdinov, "Investigating the working of text classifiers," *CoRR*, vol. abs/1801.06261, 2018.

ANNEX A

THIS PAPER WAS SUBMITTED TO JOURNAL OF INFORMATION AND SOFTWARE TECHNOLOGY IN SEPTEMBER 3, 2018.

Bug Report Severity Level Prediction in Free Libre/Open Source Software: A Survey and New Perspectives

Luiz Alberto Ferreira Gomes
Institute of Computing (IC)
University of Campinas (UNICAMP)
Campinas, Brazil
gomes.luiz@ic.unicamp.br

Ricardo da Silva Torres
Institute of Computing (IC)
University of Campinas (UNICAMP)
Campinas, Brazil
rtorres@ic.unicamp.br

Mario Lúcio Côrtes
Institute of Computing (IC)
University of Campinas (UNICAMP)
Campinas, Brazil
cortes@ic.unicamp.br

Software evolution and maintenance activities in today Free/Libre Open Source Software rely primarily on information extracted from reports registered in bug tracking systems. The severity level attribute of a bug report is considered one of the most critical variables for planning these activities. For example, this attribute 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. An incorrect assignment of severity level may generate a significant impact on the maintenance process. However, methods to assign severity level remains essentially manual with a high degree of subjectivity. Because it depends on the experience and expertise of whom have reported or reviewed the bug report, it may be considered a quite error-prone process. Due to its evident importance, both business and academic community have demonstrated a great interest in this topic, associated with an extensive investigation towards the proposal of to provide methods to automate the bug report severity prediction. This paper aims to provide a comprehensive 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. 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. Furthermore, the results presented in this review demonstrate that most of the surveyed papers proposed methods to predict severity level which extract features from unstructured text information. At the same time, these results showed that traditional machine learning algorithms and text mining methods have been playing a central role in performed experiments. However, the review also suggests that there is room for improving prediction results using state-of-the-art machine learning and text mining algorithms and techniques.

Index Terms

Software maintenance, bug tracking systems, bug reports, severity level prediction, software repositories, systematic mapping, machine learning.

I. Introduction

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

A user interacts with a BTS often through a simple mechanism called bug report form, which enables to communicate a bug to those in charge of developing or maintaining the software system [2]. Initially, he or she should inform a short description, a long description, and an associated severity level (e.g., blocker, critical, major, minor, and trivial). Subsequently, a software team member reviews this bug report and confirms or declines it (e.g., due to bug report duplication). If the bug report is confirmed, the team member should provide more information to complement the bug report form, for example, by indicating its priority and by assigning a person who will be responsible for fixing the bug.

The information of severity level is recognized as a critical variable for prioritizing and planing 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. 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 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 systematic mapping 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 systematic mapping 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 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.

The remaining of this paper is organized as follows: Section II provides the basic information background necessary to understand the research area. Section III presents related work. Section IV describes our research method. Section V presents our results. Section VI presents final findings and discussion. Finally, Section VII concludes the paper.

II. 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), and Statistical Tests. More specific concepts will be detailed as they are cited in the document.

A. Bug Tracking System (BTS)

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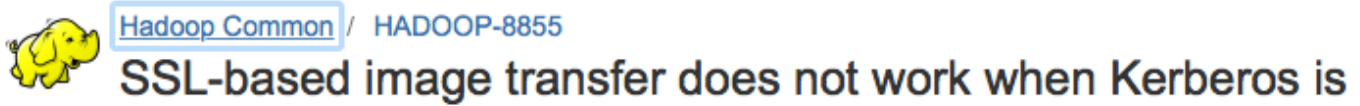
1) Bug Report: Although there is no agreement on the terminology or the amount of information that users must provide to fill a typical bug report (illustrated in Figure 1, the example refers to a bug report extracted from Bugzilla of Eclipse project.), they often describe their needs in popular BTS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

TABLE I: 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).

After the user has reported a bug, the development team is in charge of its assessment. The assessment consists of approving or, for some reason (e.g., duplication), not approving the bug. In case of approval, this team may provide complementary information by, for example, assigning a person to be responsible for handling this request or defining a new severity level for the report. Typically, the sequence of steps a bug report goes through is modeled as a state machine. Figure 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 *Unconfirmed*. The developer team will change the bug report status to *Resolved*, if the bug were not confirmed, or, otherwise, to *New*. When someone was charged to fix the bug, the bug report status will be changed to *Assigned* by the developer team. Therefore, in the standard flow, the bug report status will be assigned to resolved (bug fixed), then verified (bug checked), and finally closed. As shown in Figure 2a, others status 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.

Fig. 1: A typical bug report (https://issues.apache.org/jira/browse/HADOOP-8855, as of September 2, 2018)

Both users and development team members can define or redefine the severity level for a bug during the lifecycle. Figure 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].

B. Machine Learning

Machine Learning (ML) [11] is an application of Artificial Intelligence (AI) that provides systems the ability to learn and improve from experience without being explicitly programmed. There are two types of ML algorithms: predictive (or supervised) and descriptive (or unsupervised). A predictive algorithm builds a model based on historical training data and uses this model to predict, from the values of input attributes, an output label (class attribute) for a new sample. A predictive task is called classification when the label value is discrete, or regression when the label value is continuous.

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.

1) ML algorithms: A ML algorithm works over a dataset, which contains many samples or instances x_i, where $i = \{1..n\}$. Each instance is composed of $\{x_{i1}, x_{i2}, ..., x_{id}\}$ input attributes or independent variables, where $d = \{1..m\}$, and one output attribute or dependent variable, $x_{i(m+1)}$. Input attributes are commonly named features or feature vector, and output attribute as commonly named class or category. 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
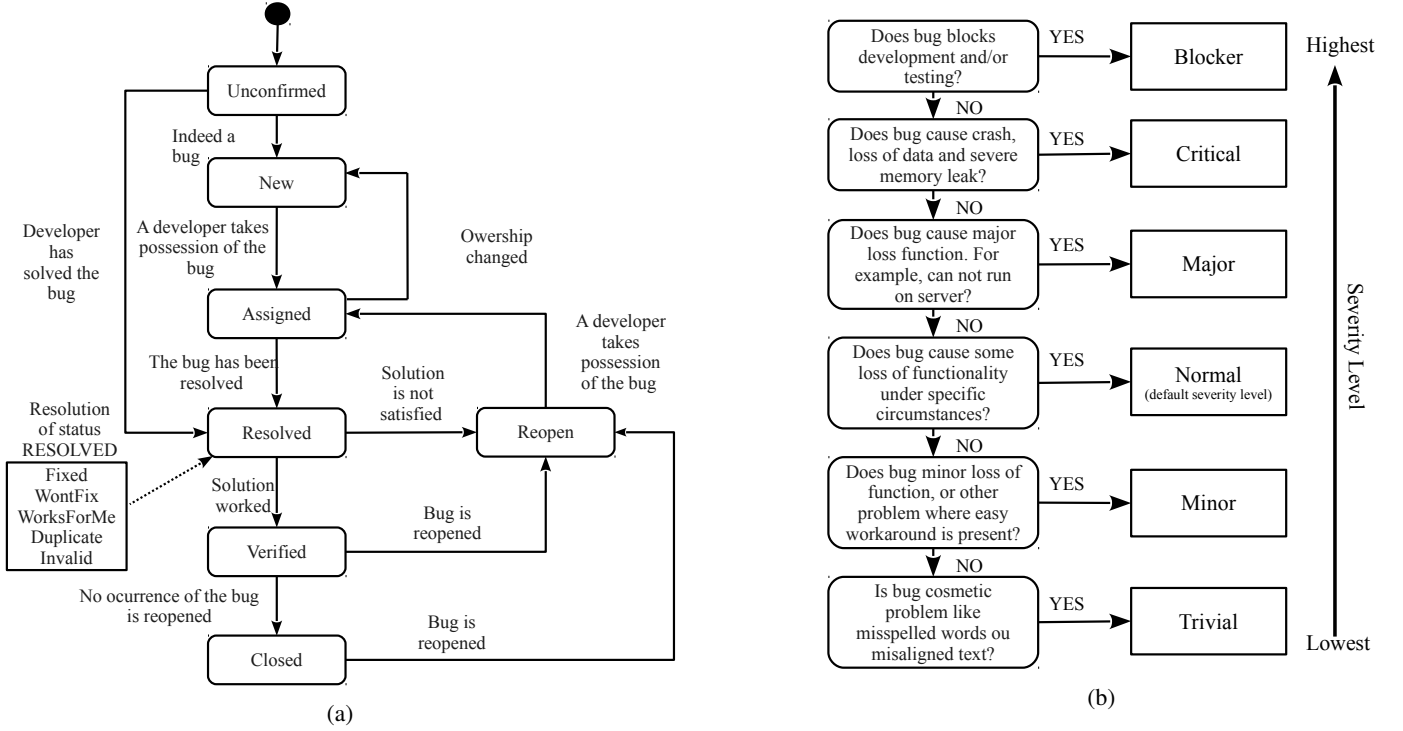
Fig. 2: (a) The bug report lifecycle according to Zhang et al. [9]. (b) Bugzilla guideline for bug report severity level assignment (https://www.bugzilla.org, as of September 2, 2018).

classification and regression tasks. However, this mapping review regards them only in the classification scenario. Next, a brief description of each one is presented [12]:

- **k-Nearest Neighbors** (k-NN) 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 points of an n-dimensional space.

- **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.

- **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 [13]. 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 [14].

- **Support Vector Machine(SVM)** represents each feature vector of as instance as a point in an n-dimensional space. The algorithm divides every point belonging to a category in a particular space. Each space is separated by a gap that is as wide as possible [12]. In Figure 3, for example, SVM will search for support vectors which are instances located at the edge of an area in space. Such vectors define boundaries between one of these class instances (e.g., the crosses) and instances from another class (e.g., the circles). . New instances are then mapped into that same space and predicted to belong to a category based on which side they are.

- **Random Forest** [15] 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.

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 [16]:

- **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
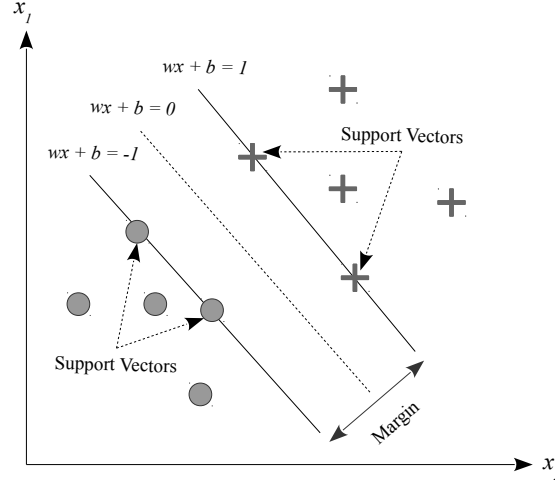
Fig. 3: A SVM model example.

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.

3) Evaluation measures: Accuracy, precision, recall, and F-measure are four measures commonly used to evaluate the performance of prediction models [17]. The computation of the values of these measures are based on a *confusion matrix* [18], which represents the number of true/false positives, and the number of true/false negatives for each instance class value when making a prediction. Each measure is described next [19]:

- **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 [17]. 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 [19] 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.

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 [19]. 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 Facelli et al. [20] 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.

5) Statistical tests: Many scenarios require running several ML algorithms to chose the best predictive model. However, when comparing two or more algorithms, statistical variations of evaluation measures may not be significant, and deciding which one is better may be difficult. In this situation, conducting statistical tests is a recommended practice for reliable comparison between predictive models under investigation [20]. A statistical test aims to gather evidence about which measure values represent a general behavior of ML algorithm [19]. A brief description of four common statistical tests is provided:

- **T-Test** [19] 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** [21] 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** [22] 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** [22] is a parametric statistical hypothesis test that can be used to test whether a sample $x_1, ..., x_n$ came from a normal distribution of the population.

C. 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 [17]. It is composed of three primary activities [18]:

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

Two of the most traditional ways of representing a document relies on the use of a bag of words (unigrams) or a bag of bigrams (when two terms appear consecutively, one after the other) [17]. In this approach all terms represent features, and thus the dimension of the feature space is equal to the number of different terms in all documents (bug reports). Methods for assigning weights to features may vary. The simplest one is to assign binary values representing the presence or absence of the term in each text. Term Frequency (TF), another type of quantification scheme, considers the number of times in which the term appears in each document. Term Frequency-Inverse Document Frequency (TD-IDF), a more complex type of scheme, takes into account the frequencies of the term in each document, and in the whole collection. The importance of a term in this scheme is proportional to its frequency in the document and inversely proportional to the frequency of the term in the collection.

III. RELATED WORK

To the extent of our knowledge, only two papers [1], [6] have reviewed the literature about 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. [23]), 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 Naïve Bayes Multinomial (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 categorizes 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. In contrast, the current mapping review investigated in depth 27 papers about bug report severity prediction published from 2010 to 2017.

Although these reviews present relevant results for research in this area, they have one difference to our work. There is no explicit focus on bug report severity prediction. In fact, those papers perform a brief review of this topic by addressing mainly bug report priority prediction. The current review has a broader goal of mapping studies addressing many research questions and concepts related to bug report severity prediction. Furthermore, this review considers relevant aspects (e.g., sampling techniques and statistical tests), not addressed before, in the characterization of papers.

IV. 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], Brhel et al. [24], Souza et al. [25], 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.

A. Research questions

The main goal of this mapping 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 or information retrieval methods for bug report severity prediction?** This research question indicates the main text mining approaches 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).

B. Paper selection

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

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 II exhibits terms and final search string used in this systematic mapping review.

TABLE II: 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"

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

TABLE III: 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

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.

EC$_1$. The paper does not have an abstract;
EC$_2$. The paper is just published as an abstract;
EC$_3$. The paper is written in a language other than English;
EC$_4$. The paper is not a primary paper (e.g., keynotes);
EC$_5$. The paper is not accessible on the Web;

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 systematic mapping review.

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.

C. Data extraction and synthesis

Figure 4 illustrates the four-stage selection process carried out in the current mapping 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 II. 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 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 IV 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.
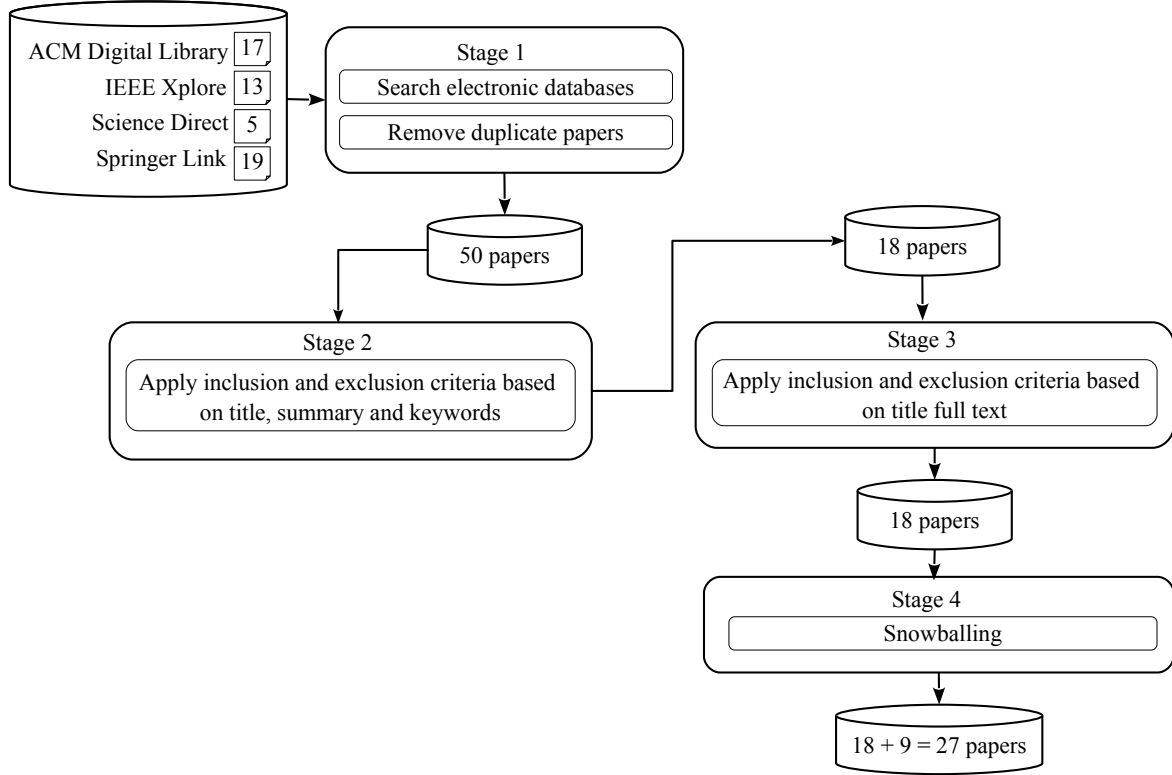
Fig. 4: Diagram of the four-stage paper selection process.

D. Limitations of this mapping

To interpret the implications of our results adequately, one needs to consider the following common limitations of all mapping reviews:

- **Mapping review completeness can never be guaranteed [24]**: 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.

TABLE IV: Selected papers

ID	Bibliographic reference
[4]	A. Lamkanfi, S. Demeyer, E. Giger, and B. Goethals, "Predicting the severity of a reported bug," in *2010 7th IEEE Working Conference on Mining Software Repositories (MSR 2010)*, May 2010, pp. 1–10. doi:10.1109/MSR.2010.5463284.
[23]	A. Lamkanfi, S. Demeyer, Q. D. Soetens, and T. Verdonck, "Comparing mining algorithms for predicting the severity of a reported bug," in *2011 15th European Conference on Software Maintenance and Reengineering*, March 2011, pp. 249–258. doi:10.1109/CSMR.2011.31.
[3]	Y. Tian, D. Lo, and C. Sun, "Information retrieval based nearest neighbor classification for fine-grained bug severity prediction," in *2012 19th Working Conference on Reverse Engineering*, Oct 2012, pp. 215–224. doi:10.1109/WCRE.2012.31.
[26]	C. Z. Yang, C. C. Hou, W. C. Kao, and 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, Dec 2012, pp. 240–249. doi:10.1109/APSEC.2012.144.
[27]	K. Chaturvedi and 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)*, vol. 4, no. 2, pp. 32–59, 2012. doi:10.4018/jossp.2012040103.
[28]	C. Z. Yang, K. Y. Chen, W. C. Kao, and 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*, June 2014, pp. 216–219. doi:10.1109/ICSESS.2014.6933548.
[29]	G. Yang, T. Zhang, and 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*, July 2014, pp. 97–106. doi:10.1109/COMPSAC.2014.16.
[10]	H. Valdivia Garcia and E. Shihab, "Characterizing and predicting blocking bugs in open source projects," in *Proceedings of the 11th Working Conference on Mining Software Repositories*, ser. MSR 2014. New York, NY, USA: ACM, 2014, pp. 72–81. doi:10.1145/2597073.2597099
[30]	M. Sharma, M. Kumari, R. K. Singh, and V. B. Singh, "Multiattribute based machine learning models for severity prediction in cross project context," in *Computational Science and Its Applications – ICCSA 2014*, B. Murgante, S. Misra, A. M. A. C. Rocha, C. Torre, J. G. Rocha, M. I. Falcão, D. Taniar, B. O. Apduhan, and O. Gervasi, Eds. Cham: Springer International Publishing, 2014, pp. 227–241. doi:10.1007/978-3-319-09156-3_17
[31]	N. K. S. Roy and B. Rossi, "Towards an improvement of bug severity classification," in *2014 40th EUROMICRO Conference on Software Engineering and Advanced Applications*, Aug 2014, pp. 269–276. doi:10.1109/SEAA.2014.51
[32]	R. K. Saha, J. Lawall, S. Khurshid, and D. E. Perry, "Are these bugs really "normal"?" in *2015 IEEE/ACM 12th Working Conference on Mining Software Repositories*, May 2015, pp. 258–268. doi:10.1109/MSR.2015.31
[9]	T. Zhang, G. Yang, B. Lee, and A. T. S. Chan, "Predicting severity of bug report by mining bug repository with concept profile," in *Proceedings of the 30th Annual ACM Symposium on Applied Computing*, ser. SAC '15. New York, NY, USA: ACM, 2015, pp. 1553–1558. doi:10.1145/2695664.2695872
[33]	G. Sharma, S. Sharma, and S. Gujral, "A novel way of assessing software bug severity using dictionary of critical terms," *Procedia Computer Science*, vol. 70, pp. 632 – 639, 2015, proceedings of the 4th International Conference on Eco-friendly Computing and Communication Systems. doi:10.1016/j.procs.2015.10.059
[34]	X. Xia, D. Lo, E. Shihab, X. Wang, and X. Yang, "Elblocker: Predicting blocking bugs with ensemble imbalance learning," *Information and Software Technology*, vol. 61, pp. 93 – 106, 2015. doi:10.1016/j.infsof.2014.12.006
[35]	S. Gujral, G. Sharma, S. Sharma, and Diksha, "Classifying bug severity using dictionary based approach," in *2015 International Conference on Futuristic Trends on Computational Analysis and Knowledge Management (ABLAZE)*, Feb 2015, pp. 599–602. doi:10.1109/ABLAZE.2015.7154933
[36]	M. N. Pushpalatha and M. Mrunalini, "Predicting the severity of bug reports using classification algorithms," in *2016 International Conference on Circuits, Controls, Communications and Computing (I4C)*, Oct 2016, pp. 1–4. doi:10.1109/CIMCA.2016.8053276
[37]	A. F. Otoom, D. Al-Shdaifat, M. Hammad, and E. E. Abdallah, "Severity prediction of software bugs," in *2016 7th International Conference on Information and Communication Systems (ICICS)*, April 2016, pp. 92–95. doi:10.1109/IACS.2016.7476092
[38]	K. K. Sabor, M. Hamdaqa, and 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*, ser. CASCON '16. Riverton, NJ, USA: IBM Corp., 2016, pp. 96–105.
[39]	Y. Tian, N. Ali, D. Lo, and A. E. Hassan, "On the unreliability of bug severity data," *Empirical Softw. Engg.*, vol. 21, no. 6, pp. 2298–2323, dec 2016. doi:10.1007/s10664-015-9409-1
[40]	T. Zhang, J. Chen, G. Yang, B. Lee, and X. Luo, "Towards more accurate severity prediction and fixer recommendation of software bugs," *Journal of Systems and Software*, vol. 117, pp. 166 – 184, 2016. doi:10.1016/j.jss.2016.02.034
[41]	K. Jin, A. Dashbalbar, G. Yang, J.-W. Lee, and B. Lee, "Bug severity prediction by classifying normal bugs with text and meta-field information," *Advanced Science and Technology Letters*, vol. 129, pp. 19–24, 2016. doi:10.14257/astl.2016.129.05
[42]	T. Choeikiwong and 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.
[43]	K. Jin, A. Dashbalbar, G. Yang, and B. Lee, "Improving predictions about bug severity by utilizing bugs classified as normal," *Contemporary Engineering Science*, vol. 9, pp. 933–942, 2016. doi:10.12988/ces.2016.6695
[44]	K. Jin, E. C. Lee, A. Dashbalbar, J. Lee, and B. Lee, "Utilizing feature based classification and textual information of bug reports for severity prediction," *Information*, vol. 19, no. 2, pp. 651–659, Feb 2016.
[5]	G. Yang, S. Baek, J.-W. Lee, and B. Lee, "Analyzing emotion words to predict severity of software bugs: A case study of open source projects," in *Proceedings of the Symposium on Applied Computing*, ser. SAC '17. New York, NY, USA: ACM, 2017, pp. 1280–1287. doi:10.1145/3019612.3019788
[45]	V. B. Singh, S. Misra, and M. Sharma, "Bug severity assessment in cross project context and identifying training candidates," *Journal of Information and Knowledge Management*, vol. 16, no. 01, p. 1750005, 2017. doi:10.1142/S0219649217500058
[46]	N. K. S. Roy and 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)*, Aug 2017, pp. 426–429. doi:10.1109/SEAA.2017.71

- **Terminology problems in search string may lead to miss some primary studies [25]**: 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 [24]**. Explicit criteria have been defined for assessing the relevance of selected papers IV-B. 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.

E. 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 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.

1) FLOSS software type (RQ_2): This scheme organizes FLOSS projects by software type. Based on the taxonomy suggested by Pressman [47], 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.

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].

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) 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 [48]:

    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.

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 systematic mapping review according to [17]:

    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.

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. [20]:

    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.

7) Evaluation measures categories (RQ_8): This scheme organizes evaluation measures used in selected papers by categories. According to Japkowicz and Shah [19], 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.

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 [19] include:

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

    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.

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 review according to Japkowicz and Shah [19] 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

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.

V. Results

This section summarizes the results of this mapping 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 IV.

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

Figure 5a shows that research on bug report severity prediction in FLOSS is recent and active with a vast number of papers (22 out of 27 $\approx$ 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 V).

TABLE V: Papers publication sources.

Paper source	Type	Reference
ACM Symposium on Applied Computing	Conference	[9]
Advanced Science and Technology Letters & Journal	Journal	[41]
Asia-Pacific Software Engineering Conference	Conference	[26]
Computational Science and Its Applications (ICCSA)	Conference	[30]
Computer Software and Applications Conference	Conference	[29]
Contemporary Engineering Sciences	Journal	[43]
Empirical Software Engineering	Journal	[39]
Futuristic Trends on Computational Analysis and Knowledge Management (ABLAZE)	Conference	[35]
Information and Software Technology	Journal	[34]
International Conference on Circuits, Controls, Communications and Computing (I4C)	Conference	[36]
International Conference on Computer Science and Software Engineering	Conference	[38]
International Conference on Information and Communication Systems (ICICS)	Conference	[37]
International Conference on Software Engineering and Service Science	Conference	[28]
International Information Institute	Journal	[44]
International Journal of Open Source Software and Process(IJOSSP)	Journal	[27]
International MultiConference of Engineers and Computer Scientists	Conference	[42]
Journal of Information & Knowledge Management	Journal	[45]
Journal of Systems and Software	Journal	[40]
Procedia Computer Science	Journal	[33]
Software Engineering and Advanced Applications (SEAA)	Conference	[31], [46]
Symposium on Applied Computing (SAC)	Conference	[5]
Working Conference on Mining Software Repositories (MSR)	Conference	[4], [23], [10], [32]
Working Conference on Reverse Engineering	Conference	[3]

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

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

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

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

Table VIII presents that is about the same papers ($\approx$ 52%) addressed the bug report severity prediction as a coarse-grained problem and $\approx$ 48% of papers as a fine-grained problem. Figure 6a shows that most papers (10 out of 27 $\approx$ 37%) addressed the severity prediction as a problem of SNS category. Moreover, 8 out of 27 ($\approx$ 20%) papers addressed it as a problem of the MC category and 5 out of 27 ($\approx$ 18.5%) addressed a problem of the MCWD category. A few papers addressed a BNB (2 out of 27 $\approx$ 7%) or an SNSWD (2 out of 27 $\approx$ 7%) problem. No paper addressed more than one problem category.

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

Table IX 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 X, most used features were *summary* ($\approx$ 74%) and *description* ($\approx$ 70.37%). Besides, a significant number of papers also reported using the features *product* ($\approx$ 37%) and *component* ($\approx$ 33%). Regarding data type of the features,

---

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

TABLE VI: Paper distribution by FLOSS.

Project	References	Total
Eclipse	[4], [23], [3], [26], [27], [29], [28], [10], [31], [32], [9], [33], [34], [35], [37], [38], [39], [40], [41], [42], [43], [44], [5], [45], [46], [23], [3], [26], [27], [29], [28], [10], [31], [32], [9], [33], [34], [35], [37], [38], [39], [40], [41], [42], [43], [44], [5], [45], [46]	25
Mozilla	[4], [3], [26], [27], [29], [10], [30], [31], [9], [34], [36], [37], [39], [40], [41], [43], [44], [45], [46]	19
Openoffice	[3], [10], [34], [39], [40]	5
Netbeans	[29], [10], [34], [40]	4
Gnome	[4], [23], [27]	3
Chromium	[10], [34]	2
Freedesktop	[10], [34]	2
GCC	[40]	1
Hibernate	[46]	1
Spring	[46]	1
Android	[5]	1
JBoss	[5]	1
Mongo-db	[46]	1
WineHQ	[36]	1

TABLE VII: FLOSS investigated in papers.

Project	Description	Category	BTS	URL (As of September 2, 2018)
Android	Mobile operating system	System software	Google	https://issuetracker.google.com
Chromium	Web browser	Application software	Google	https://www.chromium.org/issue-tracking
Eclipse	Integrated Development Environment(IDE)	Programming tool	Bugzilla	https://bugs.eclipse.org/bugs/
FreeDesktop	Base platform for desktop software on Linux and UNIX	Programing Tool	Bugzilla	https://bugs.freedesktop.org/
GCC	C compiler	Programming tool	Bugzilla	https://gcc.gnu.org/bugzilla/
Gnome	Desktop environment based on X Windows System	Application Software	Bugzilla	https://gcc.gnu.org/bugzilla/
Hibernate	Object Relation Mapper(ORM) framework	Programming tool	Jira	https://hibernate.atlassian.net
Jboss	Application server	System software	Jira	https://issues.jboss.org/s
Mongo-db	No-sql database	System software	Jira	https://jira.mongodb.org/
Mozilla	Internet tools	Application software	Bugzilla	https://bugzilla.mozilla.org/
Netbeans	Integrated Development Environment(IDE)	Programming tool	Bugzilla	https://netbeans.org/bugzilla/
OpenOffice	Office suite	Application software	Bugzilla	https://bz.apache.org/ooo/
Spring	JEE Framework	Programming tool	Jira	https://jira.spring.io
WineHQ	Compatibility layer	System software	Bugzilla	https://bugs.winehq.org/

TABLE VIII: Paper distribution by prediction problem.

Problem	References	Total
Coarse-grained	[4], [23], [26], [28], [31], [10], [33], [35], [34], [37], [41], [43], [32], [44]	14
Fine-grained	[3], [27], [29], [30], [9], [36], [38], [39], [40], [42], [45], [5], [46]	13

unstructured text (26 out of 27 ≈ 96%) was most used data type in the papers (Figure 6b). Only one paper (Meera et al. [30]) reported the use of features belonging to each data type presented in the chart.

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

Table XI 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.

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

As shown in Table XII, ≈ 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:
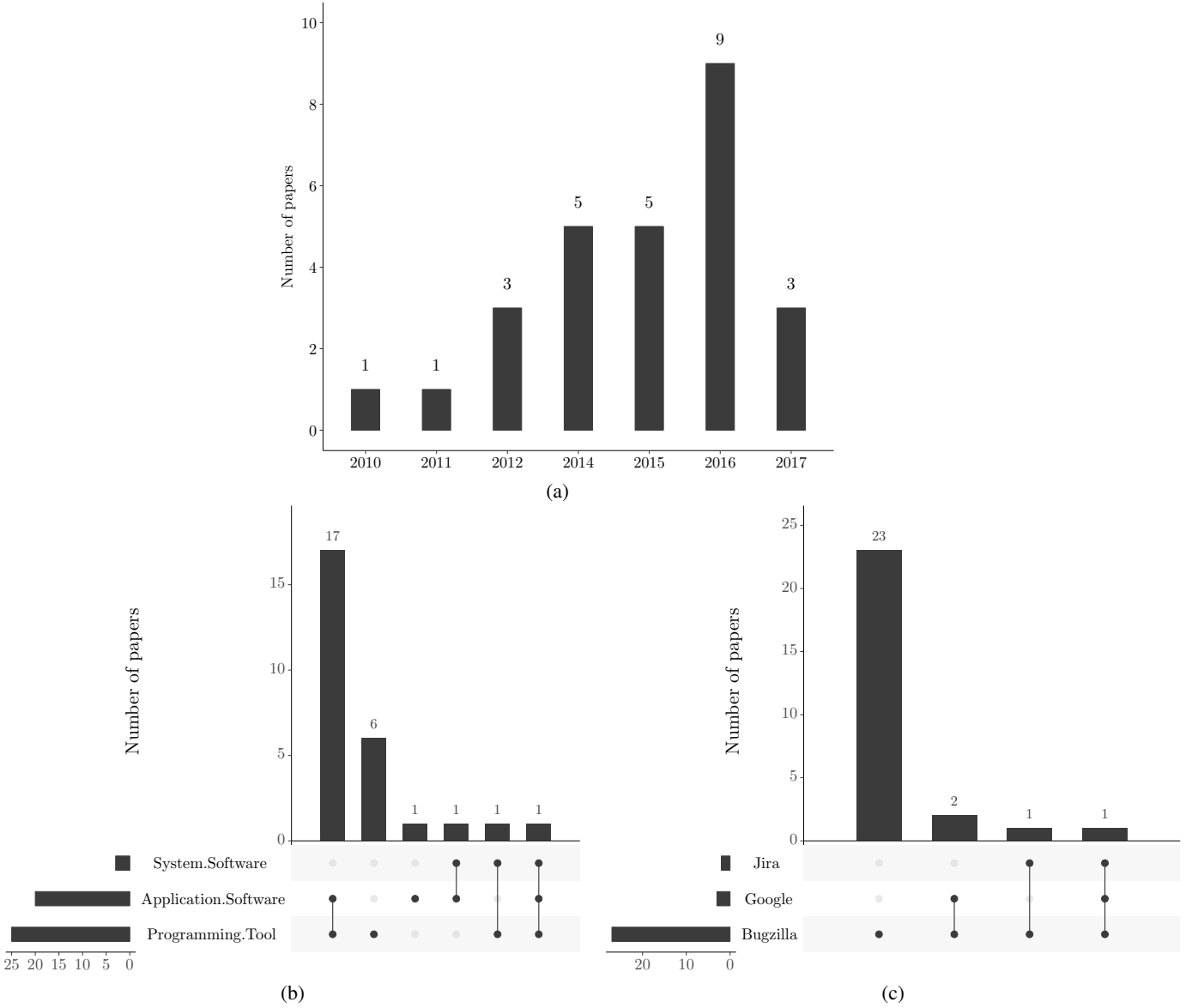
Fig. 5: (a) Paper distribution by year (67% published in conferences and 33% in journals), (b) paper distribution by FLOSS (c) paper distribution by BTS.

one paper used BM25, one paper BM25ext, and another KL divergence.

Figure 6c shows that most used text mining method (9 out of 15 $\approx$ 44%) belongs to term category. Besides, only one paper (Yang et al. [5]) reported the use of text mining belonging to each category presented in the chart.

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

Table XIII presents that most papers used k-NN and NB ($\approx$ 44%) and that NBM has also been used quite frequently ($\approx$ 40%). As shown in Figure 6d, 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.

[2]Topic Model [40] 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 ou more topics.

[3]BM25ext [40] is a 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.

[4]Kullback-Leibler(KL) [29] divergence measures the difference between two probability over the same variable.

TABLE IX: Feature descriptions.

Feature	Description	Category	Calculated?	Bugzilla	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[1]	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

[5]Näive Bayes Multinomial (NBM) [23] is similar to Näive Bayes. However, the output class is not only determined by presence or absence of term, but NBM also uses the number of occurrences of the terms to decide.

[6]C4.5 [10] is an algorithm used to generate a decision tree which follows a greedy divide and conquer strategy in training step.

[7]Bagging Ensemble [34] 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.

[8]AdaBoost [37] is an ensemble algorithm that attempts to produce a very accurate classification rule by combining moderately inaccurate weak classifiers.

[9]Functional Tree [37] is a classification tree that could have logistic regression function at the inner nodes and or leaves.

[10]Random Tree [37] is an ensemble classifier that consists of directed graphs build with a random process.

[11]Radial Basis Function (RBF) Neural Network [12] 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.

[12]RIPPER [49] 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.

[13]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 X: Paper distribution by features.

Feature	References	Total
Summary	[4], [23], [3], [26], [27], [29], [28], [30], [32], [33], [36], [37], [39], [40], [41], [43], [44], [5], [45], [46]	20
Description	[4], [3], [29], [28], [10], [31], [9], [34], [35], [36], [37], [39], [38], [40], [41], [43], [44], [5], [46]	19
Product	[3], [29], [10], [34], [36], [39], [40], [41], [43], [44]	10
Component	[3], [29], [10], [34], [36], [40], [41], [43], [44]	9
Severity	[10], [34], [41], [43], [44]	5
Priority	[29], [10], [30], [34]	4
Reporter	[41], [43], [44]	3
Description size	[10], [34]	2
Number in CC list	[10], [34]	2
Reporter blocking experience	[10], [34]	2
Reporter experience	[10], [34]	2
Reporter name	[10], [34]	2
Attachments	[28]	1
Bug fix time	[30]	1
Bug Id	[42]	1
CC list	[30]	1
Comment	[10]	1
Comment size	[34]	1
Complexity	[30]	1
Importance	[36]	1
Number of comments	[30]	1
Number of dependents	[30]	1
Number of duplicates	[30]	1
Platform	[34]	1
Report length	[28]	1
Summary weight	[30]	1

TABLE XI: Paper distribution by feature selection methods.

Method	Category	References	Total
Information Gain	Filter	[26], [27], [33], [39], [46]	5
Chi-Square	Filter	[26], [31], [33]	3
Correlation Coefficient	Filter	[26]	1

H. 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 XIV): 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 Figure 7a. Only one paper (Roy et al. [31]) investigated all four categories of measures presented in the chart.

[14]Mean Reciprocal Rank (MRR) [14] 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.

[15]Effectiveness Ratio [34] is a cost-effectiveness measure, which evaluates prediction performance given a cost limit.

[16]Krippendorff's Alpha Coefficient [39] is a statistical measure of the agreement achieved when coding a set o units analysis of values of a variable.

TABLE XII: Paper distribution by text mining models.

Technique	Category	References	Total
Vector Extraction Models			
Unigrams	Character	[4], [23], [3], [26], [27], [29], [28], [31], [33], [35], [37], [38], [39], [40], [41], [42], [43], [44], [5], [45], [46]	20
Bigrams	Character	[3], [31], [39], [40]	4
Topic model[2]	Concept	[29], [9], [40]	3
Vector Weighting Models			
TF-IDF	Term	[23], [27], [31], [9], [33], [35], [38], [45], [46]	9
TF	Term	[23], [26], [37]	3
Binary	Term	[23]	1
Vector Similarity/Dissimilarity Functions			
BM25ext[3]	Term	[40]	1
BM25	Term	[3]	1
KL divergence[4]	Term	[29]	1

TABLE XIII: Papers distribution by ML algorithms

Algorithm	Category	References	Total
KNN	Distance	[23], [3], [29], [10], [30], [9], [35], [38], [39], [40], [42], [45]	12
NB	Probabilistic	[4], [23], [27], [29], [10], [30], [31], [32], [9], [37], [42], [45]	12
NBM[5]	Probabilistic	[23], [26], [28], [9], [33], [35], [39], [41], [43], [44], [5]	11
SVM	Otimization	[23], [27], [30], [39], [42], [45], [46]	7
Random Forest	Ensemble	[27], [10], [34], [37], [42]	5
C4.5[6]	Searching	[27], [10], [36]	3
Bagging	Ensemble	[34], [36]	2
Decision Tree	Searching	[39], [42]	2
AdaBoost	Ensemble	[37]	1
Functional Tree	Searching	[37]	1
Random Tree	Ensemble	[37]	1
RBF Networks	Optimization	[37]	1
RIPPER[7]	Other	[27]	1
Zero-R[8]	Other	[10]	1

TABLE XIV: Papers distribution by evaluation measures.

| Measure	Category	References	Total
Precision	Single Class	[4], [3], [27], [29], [10], [30], [31], [9], [33], [34], [35], [36], [38], [40], [41], [42], [43], [44], [5], [45], [46]	21
Recall	Single Class	[4], [3], [27], [29], [10], [30], [31], [9], [34], [36], [38], [40], [41], [42], [43], [44], [5], [45], [46]	19
F-measure	Single Class	[4], [3], [27], [29], [10], [30], [31], [9], [34], [38], [40], [41], [42], [43], [44], [5], [45], [46]	18
Accuracy	Multi-Class	[27], [29], [10], [30], [31], [32], [33], [35], [36], [37], [45]	11
AUC	Summary	[4], [23], [26], [28], [31]	5
ROC	Graphical	[4], [31]	2
MRR[9]	Single Class	[29], [40]	2
Effectiveness Ratio[10]	Single Class	[34]	1
Krippendorff's Alpha Reliability[11]	Single Class	[39]	1
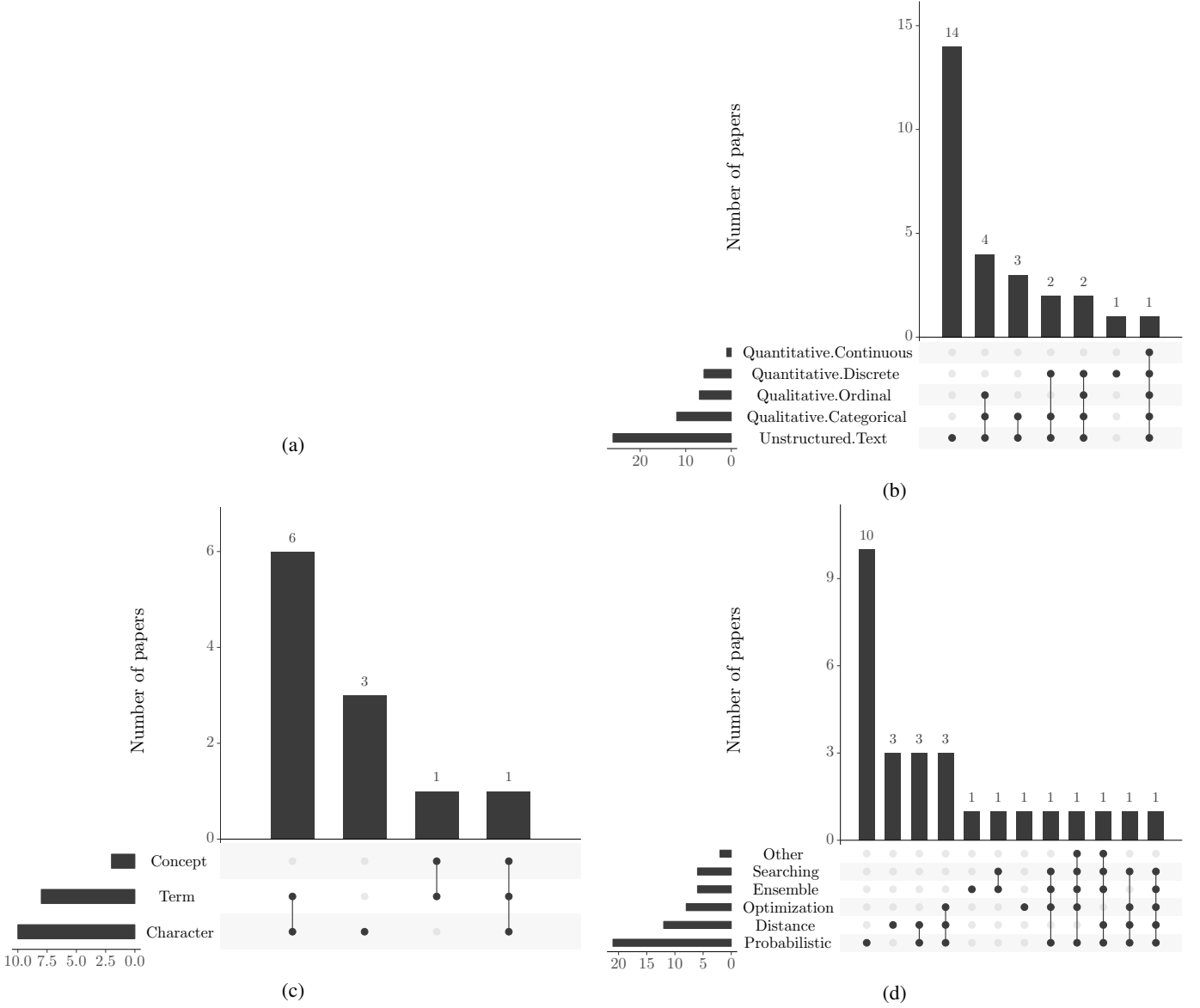
Fig. 6: (a) Paper distribution by prediction problem category, (b) paper distribution by feature data category, (c) paper distribution by text mining method category (d) paper distribution by ML algorithm categories.

I. 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 XV applied 10-Fold CV ($\approx 66\%$) to generate more reliable effectiveness estimates. Figure 7b 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. [37]) investigated both non-resampling and simple resampling methods for bug report severity prediction.

TABLE XV: Paper distribution by sampling methods.

Method	Category	References	Total
10-Fold CV	Simple Re-sampling	[23], [26], [27], [28], [34], [36], [37], [43], [5], [46]	10
Stratified 10-Fold CV[12]	Simple Re-sampling	[10], [31], [45]	3
SMOTE[13]	Simple Re-sampling	[34], [42]	2
Hold-out	No Re-sampling	[37]	1
03-Fold CV	Simple Re-sampling	[42]	1
05-Fold CV	Simple Re-sampling	[33]	1

J. 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 XVI, most of the papers used Wilcoxon Signed Rank Test (8 out of 27 ≈ 29%) and t-test (7 out of 27 - ≈ 26%) to compare the performance of ML algorithms. Figure 7c calls attention that only ten papers (≈ 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 ≈ 80%) applied a non-parametric test, and 5 out of 10 (50%) papers employed both parametric and non-parametric tests.

TABLE XVI: Paper distribution by statistical tests.

Test	Category	References	Total
Wilcoxon signed rank test	Non-parametric	[29], [10], [9], [34], [40], [43], [5], [45]	8
T-test	Parametric	[29], [9], [40], [42], [43], [5], [46]	7
Proportion test	Parametric	[46]	1
Shapiro-Wilk test	Parametric	[5]	1

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

Most of the papers listed in Table XVII used RapidMiner (5 out of 18 ≈ 27%) to run their experiments. However, Figure 7d 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.

TABLE XVII: Paper distribution by software tools.

Tool	Category	References	Total
RapidMiner	CSS	[27], [30], [33], [35], [45]	5
WEKA	FLOSS	[23], [34], [36], [39]	4
R	FLOSS	[29], [9], [5]	3
NLTK	FLOSS	[31], [40], [43]	3
Statisca	CSS	[27], [30]	2
TMT	FLOSS	[29], [40]	2
OpenNLP	FLOSS	[3]	1
WordNet	FLOSS	[31]	1
Ruby	FLOSS	[4]	1
WVTool	FLOSS	[26]	1
CoreNLP	FLOSS	[5]	1

L. 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 XVIII). Just one paper (Tian et al. [3]) presented an online solution.

TABLE XVIII: Solution types proposed in selected papers.

Tool	References	Total
offline	[4], [23], [26], [27], [29], [28], [10], [30], [31], [32], [9], [33], [34], [35], [36], [37], [38], [39], [40], [41], [42], [43], [44], [5], [45], [46]	26
online	[3]	1

[17]Stratified k-fold CV [19] is a k-fold CV variation, which ensures that class distribution is respected in the training and testing sets created at every fold.
[18]Synthetic Minority Oversampling TEchnique (SMOTE) [50] is an sampling approach in which the minority class is over-sampled by creating "synthetic" examples rather than by over-sampling with replacement.
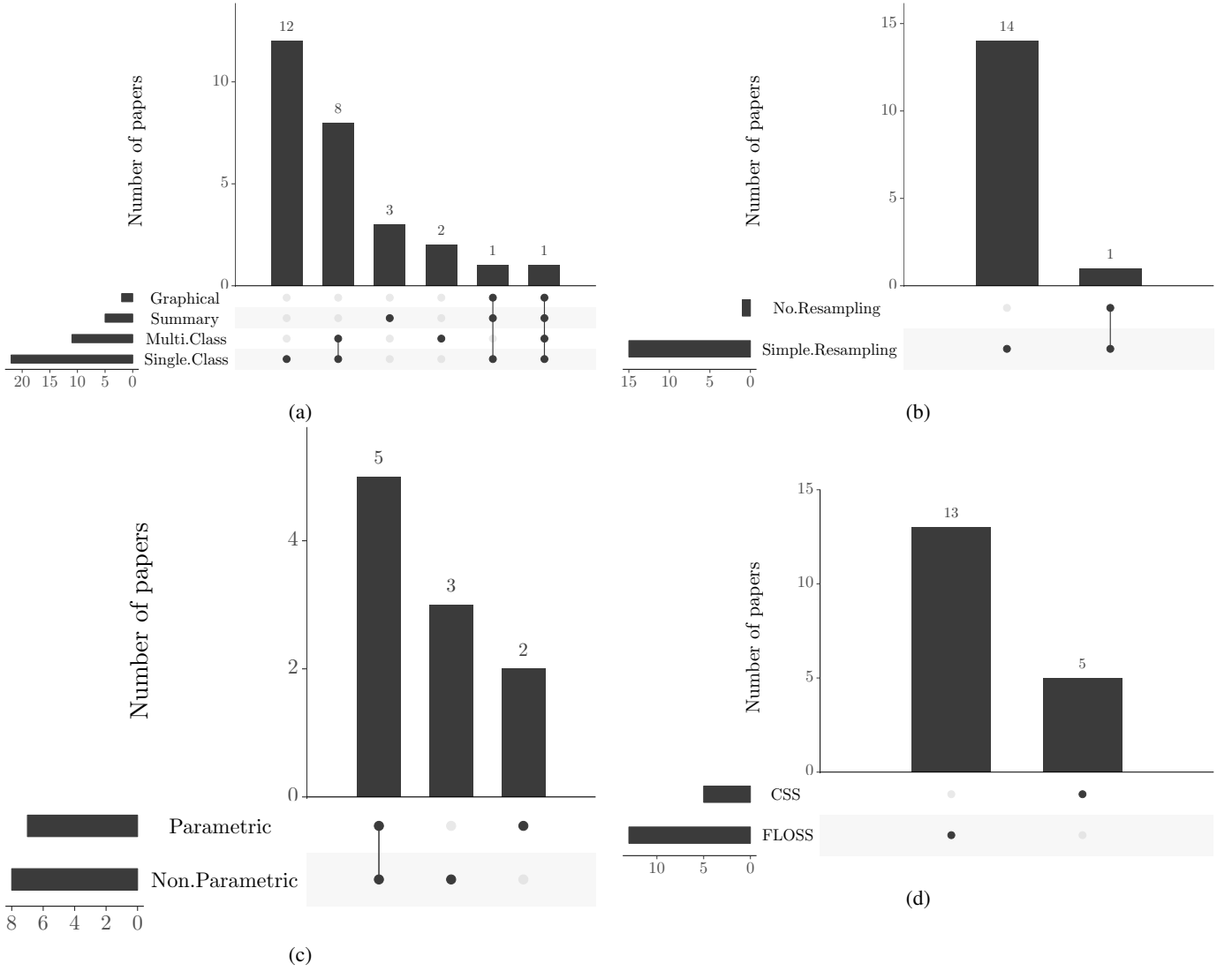
Fig. 7: (a) Paper distribution by evaluate measure categories, (b) paper distribution by sampling method categories, (c) paper distribution by statistical test categories, and (d) paper distribution by software tool categories.

VI. DISCUSSION

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

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], [23], 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[18], and the last one tracks bugs for many internal and external Google Inc. projects[19]. Surprisingly, for example, no paper investigated Github, another outstanding collaborative website which hosted more than 67 million projects (including many FLOSS)[20].

From 14 FLOSS identified in this 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] [23]. 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

---

[18]http://www.apache.org/index.html#projects-list, as of September 2, 2018.

[19]https://developers.google.com/issue-tracker/, as of September 2, 2018.

[20]https://octoverse.github.com/, as of September 2, 2018.

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 [32] 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 [12], do not work with qualitative data [11]. 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 [26], 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 [11]: (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 ming 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.

It seems that researchers have adopted a conservative approach regarding the use of ML algorithms. Most papers applied at least one of 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 [51]. This characteristic is particularly critical in bug report repositories which are intrinsically imbalanced in practice [46]. To minimize such distortion, Tian et al. [39], instead, recommend measuring performance using inter-rater agreement based metrics, such as Cohen's Kappa [52] or Krippendorff's Alpha [53]. Despite its well-known issues in an imbalanced dataset [19], 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.

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 [54]. Japkowicz et el. [19] 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 [55]. 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 [20]. Besides, no paper investigated Analysis of Variance (ANOVA), a robust ranked-based test recommended by Kitchenham et al. [55] for Empirical Sofware 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 [56] is a problem intrinsic to supervised ML algorithms, which may make it difficult to turn these offline solutions to online.

VII. Conclusions and Research Directions

A systematic mapping 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 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 [32], 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 [57]) 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 [58] seems to be a promising research direction.
- Researchers should investigate more recent techniques (e.g., continuous learning [59]) 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) [60]. 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.

References

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

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

[3] Y. Tian, D. Lo, and C. Sun, "Information retrieval based nearest neighbor classification for fine-grained bug severity prediction," in *2012 19th Working Conference on Reverse Engineering*, Oct 2012, pp. 215–224.

[4] A. Lamkanfi, S. Demeyer, E. Giger, and B. Goethals, "Predicting the severity of a reported bug," in *2010 7th IEEE Working Conference on Mining Software Repositories (MSR 2010)*, May 2010, pp. 1–10.

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

[6] J. Uddin, R. Ghazali, M. M. Deris, R. Naseem, and H. Shah, "A survey on bug prioritization," *Artificial Intelligence Review*, vol. 47, no. 2, pp. 145–180, Feb 2017.

[7] B. Kitchenham and S. Charters, "Guidelines for performing systematic literature reviews in software engineering," 2007.

[8] K. Petersen, R. Feldt, S. Mujtaba, and M. Mattsson, "Systematic mapping studies in software engineering," in *Proceedings of the 12th International Conference on Evaluation and Assessment in Software Engineering*, ser. EASE'08. Swindon, UK: BCS Learning & Development Ltd., 2008, pp. 68–77.

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

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

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

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

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

[14] J. Zhou, H. Zhang, and 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*, ser. ICSE '12. Piscataway, NJ, USA: IEEE Press, 2012, pp. 14–24.

[15] L. Breiman, "Random Forests," *Machine Learning*, vol. 45, no. 1, pp. 5–32, 2001.

[16] Z. Zheng, X. Wu, and R. Srihari, "Feature selection for text categorization on imbalanced data," *SIGKDD Explor. Newsl.*, vol. 6, no. 1, pp. 80–89, Jun. 2004.

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

[18] G. Williams, *Data Mining with Rattle and R: The Art of Excavating Data for Knowledge Discovery*, 2011.

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

[20] K. Facelli, A. C. Lorena, J. Gama, and A. C. d. Carvalho, *Inteligência Artificial: Uma Abordagem de Aprendizado de Máquina*. Rio de Janeiro, RJ, Brasil: LTC, 2015.

[21] F. Wilcoxon, *Individual Comparisons by Ranking Methods*, S. Kotz and N. L. Johnson, Eds. New York, NY: Springer New York, 1992.

[22] N. Lewis, *100 Statistical Tests In R*, ser. Easy R Series. Createspace Independent Pub, 2013.

[23] A. Lamkanfi, S. Demeyer, Q. D. Soetens, and T. Verdonck, "Comparing mining algorithms for predicting the severity of a reported bug," in *2011 15th European Conference on Software Maintenance and Reengineering*, March 2011, pp. 249–258.

[24] M. Brhel, H. Meth, A. Maedche, and K. Werder, "Exploring principles of user-centered agile software development: A literature review," *Information and Software Technology*, vol. 61, pp. 163 – 181, 2015.

[25] Érica Ferreira de Souza, R. de Almeida Falbo, and N. L. Vijaykumar, "Knowledge management initiatives in software testing: A mapping study," *Information and Software Technology*, vol. 57, pp. 378 – 391, 2015.

[26] C. Z. Yang, C. C. Hou, W. C. Kao, and 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, Dec 2012, pp. 240–249.

[27] K. Chaturvedi and 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)*, vol. 4, no. 2, pp. 32–59, 2012.

[28] C. Z. Yang, K. Y. Chen, W. C. Kao, and 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*, June 2014, pp. 216–219.

[29] G. Yang, T. Zhang, and 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*, July 2014, pp. 97–106.

[30] M. Sharma, M. Kumari, R. K. Singh, and V. B. Singh, "Multiattribute based machine learning models for severity prediction in cross project context," in *Computational Science and Its Applications – ICCSA 2014*, B. Murgante, S. Misra, A. M. A. C. Rocha, C. Torre, J. G. Rocha, M. I. Falcão, D. Taniar, B. O. Apduhan, and O. Gervasi, Eds. Cham: Springer International Publishing, 2014, pp. 227–241.

[31] N. K. S. Roy and B. Rossi, "Towards an improvement of bug severity classification," in *2014 40th EUROMICRO Conference on Software Engineering and Advanced Applications*, Aug 2014, pp. 269–276.

[32] R. K. Saha, J. Lawall, S. Khurshid, and D. E. Perry, "Are these bugs really "normal"?" in *2015 IEEE/ACM 12th Working Conference on Mining Software Repositories*, May 2015, pp. 258–268.

[33] G. Sharma, S. Sharma, and S. Gujral, "A novel way of assessing software bug severity using dictionary of critical terms," *Procedia Computer Science*, vol. 70, pp. 632 – 639, 2015, proceedings of the 4th International Conference on Eco-friendly Computing and Communication Systems.

[34] X. Xia, D. Lo, E. Shihab, X. Wang, and X. Yang, "Elblocker: Predicting blocking bugs with ensemble imbalance learning," *Information and Software Technology*, vol. 61, pp. 93 – 106, 2015.

[35] S. Gujral, G. Sharma, S. Sharma, and Diksha, "Classifying bug severity using dictionary based approach," in *2015 International Conference on Futuristic Trends on Computational Analysis and Knowledge Management (ABLAZE)*, Feb 2015, pp. 599–602.

[36] M. N. Pushpalatha and M. Mrunalini, "Predicting the severity of bug reports using classification algorithms," in *2016 International Conference on Circuits, Controls, Communications and Computing (I4C)*, Oct 2016, pp. 1–4.

[37] A. F. Otoom, D. Al-Shdaifat, M. Hammad, and E. E. Abdallah, "Severity prediction of software bugs," in *2016 7th International Conference on Information and Communication Systems (ICICS)*, April 2016, pp. 92–95.

[38] K. K. Sabor, M. Hamdaqa, and 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*, ser. CASCON '16. Riverton, NJ, USA: IBM Corp., 2016, pp. 96–105.

[39] Y. Tian, N. Ali, D. Lo, and A. E. Hassan, "On the unreliability of bug severity data," *Empirical Softw. Engg.*, vol. 21, no. 6, pp. 2298–2323, dec 2016.

[40] T. Zhang, J. Chen, G. Yang, B. Lee, and X. Luo, "Towards more accurate severity prediction and fixer recommendation of software bugs," *Journal of Systems and Software*, vol. 117, pp. 166 – 184, 2016.

[41] K. Jin, A. Dashbalbar, G. Yang, J.-W. Lee, and B. Lee, "Bug severity prediction by classifying normal bugs with text and meta-field information," *Advanced Science and Technology Letters*, vol. 129, pp. 19–24, 2016.

[42] T. Choeikiwong and 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.

[43] K. Jin, A. Dashbalbar, G. Yang, and B. Lee, "Improving predictions about bug severity by utilizing bugs classified as normal," *Contemporary Engineering Science*, vol. 9, pp. 933–942, 2016.

[44] K. Jin, E. C. Lee, A. Dashbalbar, J. Lee, and B. Lee, "Utilizing feature based classification and textual information of bug reports for severity prediction," *Information*, vol. 19, no. 2, pp. 651–659, Feb 2016.

[45] V. B. Singh, S. Misra, and M. Sharma, "Bug severity assessment in cross project context and identifying training candidates," *Journal of Information and Knowledge Management*, vol. 16, no. 01, p. 1750005, 2017.

[46] N. K. S. Roy and 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)*, Aug 2017, pp. 426–429.

[47] R. Pressman, *Software Engineering: A Practitioner's Approach*, 7th ed. New York, NY, USA: McGraw-Hill, Inc., 2010.

[48] I. Guyon and A. Elisseeff, "An introduction to variable and feature selection," *J. Mach. Learn. Res.*, vol. 3, pp. 1157–1182, Mar. 2003.

[49] T. Menzies and A. Marcus, "Automated severity assessment of software defect reports," in *2008 IEEE International Conference on Software Maintenance*, Sept 2008, pp. 346–355.

[50] N. V. Chawla, K. W. Bowyer, L. O. Hall, and W. P. Kegelmeyer, "Smote: Synthetic minority over-sampling technique," *J. Artif. Int. Res.*, vol. 16, no. 1, pp. 321–357, Jun. 2002.

[51] L. A. Jeni, J. F. Cohn, and 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*, Sept 2013, pp. 245–251.

[52] A. Ben-David, "Comparison of classification accuracy using cohen's weighted kappa," *Expert Systems with Applications*, vol. 34, no. 2, pp. 825 – 832, 2008.

[53] K. Krippendorff, "Computing krippendorff's alpha-reliability," *Departamental Papers (ASC)*.

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

[55] B. Kitchenham, L. Madeyski, D. Budgen, J. Keung, P. Brereton, S. Charters, S. Gibbs, and A. Pohthong, "Robust statistical methods for empirical software engineering," *Empirical Software Engineering*, vol. 22, no. 2, pp. 579–630, Apr 2017.

[56] A. Nigam, B. Nigam, C. Bhaisare, and 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)*, March 2012, pp. 393–397.

[57] G. Wang, C. Li, W. Wang, Y. Zhang, D. Shen, X. Zhang, R. Henao, and L. Carin, "Joint embedding of words and labels for text classification," *CoRR*, vol. abs/1805.04174, 2018.

[58] I. Goodfellow, Y. Bengio, and A. Courville, *Deep Learning*. MIT Press, 2016.

[59] Z. Chen and B. Liu, "Lifelong machine learning," *Synthesis Lectures on Artificial Intelligence and Machine Learning*, vol. 10, no. 3, pp. 1–145, 2016.

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