International Journal of Industrial Engineering
2023;7(2):43-49.
ISSN: 2456-8449

Bug Tracking System to Reduce Duplicate Bug Reports Using Cost-Aware Algorithm

G. Kavitha[1*], V. Venkataramanan[2], J Lenin[3], M. Robinson Joel[4]

[1]*Department of Computer Science and Engineering, Muthayammal Engineering College (Autonomous), Namakkal, Tamil Nadu, India*
[2]*Department of Electronics and Communication Engineering, D.J Sanghvi College of Engineering, Mumbai, Maharashtra, India*
[3]*Department of Information Technology, Al Musanna College of Technology, Muladdah, Muscat, Sultanate of Oman.*
[4]*Department of Information Technology, Kings Engineering College, Chennai, Tamil Nadu, India.*

[*]*Corresponding author: kavitha032003@yahoo.co.in*

Abstract. Software engineers rely heavily on bug-tracking solutions to help direct their maintenance efforts. In certain projects, as many as quarters of all bug reports are duplicates, reducing the usefulness of these systems. Bugs are prioritized by cost-conscious algorithms according to their effect on consumers, possible business hazards, and resource availability. By doing this, the bug tracking system can concentrate on fixing high-priority problems first, reducing the effect on users and the software system. A cost-aware algorithm in a bug tracking system aims to maximize the problem resolution process via resource allocation, bug fix prioritization, and cost management of software issues. Manually identifying duplicate bug reports is a time-consuming and expensive procedure that adds to developers' workloads. To prevent developers from wasting time, suggest a system that may automatically sort incoming problem reports into distinct categories. This method makes use of graph clustering in addition to textual semantics and surface information to make duplication predictions. Simulate a real-time bug reporting environment and conduct experiments using a dataset of 29,000 bug reports from Mozilla projects. By eliminating 8% of duplicate bug reports while ensuring that developers get at least one report for each genuine fault, the solution can cut down on development costs significantly. A bug-tracking system is a piece of software that acts as a database for recording software defects and user recommendations. The result shows that the false positive rate is reduced to 1% by using the Cost Aware algorithm.

Keywords: Bug Tracking System, Text, Filtering, Cost-Aware algorithm and Features

INTRODUCTION

The term "software testing" refers to the process of looking for and fixing flaws in software. Most errors in computer programs are the result of human error, either in the original programming or in the design of the application itself. Bugs may take several forms while using software programs, making it difficult for a single user to identify and effectively handle them. Bug-tracking systems were developed to collect, organize, and report information about an application's reported defects to address this problem [1]. Many different bug-tracking systems, both private and open-source, already exist, and others are under development to address the wide range of defects that arise as software evolves. With the constantly developing varieties of software programs, there is a need for a tool that will aid in correcting and monitoring the progress of problem repairs. In this research, five defect tracking systems (two open sources and three proprietary) are compared. New suggested defects tracking systems called "Bug-trac" consider all possible software applications and the bugs that might arise from them. When compared to current defect tracking systems, the suggested approach is seen as an improvement [2].

The input customers provide in app stores is invaluable to app developers, who can then incorporate it into future updates. However, due to the language barrier between users and developers, it might be difficult to identify user input that corresponds to current problem reports in issue trackers. In this paper, we present DeepMatcher, an automated solution that uses cutting-edge deep learning techniques to correlate feedback on apps with existing bug reports. Quantitative and qualitative tests on DeepMatcher using four free, publicly available applications are conducted. The mean hit ratio for DeepMatcher was 0.71, and the mean average precision was 0.55. DeepMatcher could not locate any duplicate bug reports for 91 different problems. Analysis of these 91 complaints and the corresponding issues in the tracked applications revealed that users had mentioned the issue 47 times before the

Received: 15.05.2023 Revised: 01.07.2023 Accepted: 20.07.2023

developers had been aware of it. Present results and explore the many applications of DeepMatcher [3]. As a result of their portability and versatility, mobile devices help the handicapped in many ways today, including facilitating the discovery of accessible sites, facilitating image and voice-based communication, and allowing for the creation of individualized user interfaces and vocabularies. The libraries that make up these accessibility frameworks are embedded directly into a wide range of programs so that their users may make use of their accessibility features. These frameworks, like any software, are prone to bugs. Software engineers document these problems as bugs. Reports on accessibility issues must be addressed immediately since they severely reduce the use of software [4].

In this setting, manually sifting through many bug reports in search of accessibility-related issues is both time-consuming and prone to human mistakes. The categorization of mobile app user evaluations has been studied before for a variety of objectives, such as the identification of bug reports, the identification of feature requests, the optimization of app performance, etc. It is challenging for software developers to prioritize and promptly fix accessibility-related problem reports since no previous study has looked at their identification. The purpose of this article is to automatically determine whether a particular bug report is related to accessibility or not in order to aid developers with this tedious procedure [5]. So, take on the challenge of categorizing accessibility issue reports as a two-way problem. Use a dataset consisting of accessibility bug reports from widely used open-source programs like Mozilla Firefox and Google Chromium to train the model. The approach is built on the idea that these reports may teach us the right discriminative characteristics, i.e., keywords, to use when representing accessibility problems. The results of stratified cross-validation evaluations of the trained model demonstrate that the classifier is quite effective, with F1-scores of 93% [6].

Defects, problems, tasks, change requests, etc., are all managed using issue trackers (like Jira) used by software engineers. In this paper, investigate (a) the linguistic representation (e.g., as adjectives) of architectural knowledge concepts (such as component behavior and contextual constraints) in issues, (b) the frequency with which these concepts appear in issues, and (c) the co-occurrence of these concepts. Looked at three major Apache projects' Jira issue trackers and studied their bug reports. Related concerns to architecturally relevant modifications in the source code of the examined systems to identify "architecturally relevant" issues. By assigning labels to a subset of problems by hand, we were able to create a code book. When I was exhausted by all possible ways of thinking about a problem, I programmed it. Research provides empirical evidence in favor of search tools that may extract architectural knowledge ideas from problems for later re-use [7]. The problem statement is discussed below. The difficulties and problems that businesses may have in effectively monitoring and resolving software defects while considering numerous criteria, such as resource allocation, prioritizing, and total cost, are addressed in the problem statement for a bug-tracking system employing a cost-aware algorithm. It may be difficult for traditional bug-tracking systems to allocate work to developers according to their availability and level of competence.

The technique helps to improve software quality by methodically identifying and fixing bugs, especially important ones. Therefore, the software program becomes more dependable and stable, and either meets or surpasses user expectations. Together, these contributions to the work provide a bug-tracking system that improves the overall efficacy and efficiency of software development projects, fits with corporate goals, and maximizes problem-resolution procedures. The following section, be li, a literature survey, is discussed in section 2. After that, the proposed system is discussed using the Cost Aware algorithm for the Bug Tracking system in section 3. Then, the results and discussion for the given dataset to reduce false positives are discussed in section 4. Finally, the conclusion provides the overall performance of the healthcare system and future work.

LITERATURE SURVEY

Misclassification while searching for and removing false positives from the bug report database is a popular topic of investigation. Costs rise when more time, energy, and upkeep are needed to triage and repair defects because of the time wasted weeding out irrelevant complaints. As a result, this is a well-researched and discussed topic here. This research suggests using classification algorithms in the bug report repository to detect non-bug submissions automatically. There are three factors considered. First, unigram and CamelCase are utilized in bug reports, with the latter being used for feature augmentation via the usage of CamelCase terms [8]. As a second step, evaluate five different word weighting systems and compare them to find the one that works best for this purpose. Finally, the key techniques for modeling non-bug reports identifiers are members of the support vector machines (SVM) family, such as binary-class SVM, one-class SVM based on Schoellkopf methodology, and support vector data descriptions (SVDD). The results of the recall, precision, and F1 tests show that the bug reports repository can effectively filter out false positives. When compared to earlier, well-known research, findings may

be acceptable; when comparing the Schoellkopf technique and SVDD methods, the performances of non-bug report IDs using a weighting scheme produced the best value [9].

Bug tracking systems are used in enterprise-level software development settings, where defect reports are tracked and reviewed by subject matter experts. Users with wildly varying writing styles and habits may submit bug reports that each describe the same issues in slightly different wording. As a result, reports of the same fault might have vastly different descriptions, leading to non-trivial duplication. An expert must review all incoming reports while attempting to designate probable duplicates to prevent unnecessary labor for the development team. However, this method is neither simple nor scalable, and it has an obvious effect on the amount of time required to address bugs. Deep neural techniques that consider the hybrid representation of bug reports, making use of both structured and unstructured data, have been the primary focus of recent attempts to discover duplicate bug reports [10]. Unfortunately, these methods fail to consider the fact that a single problem may have several copies that have already been recognized, each with its own set of details, such as textual description, titles, and categories. This paper offers SiameseQAT, a technique for identifying duplicate bug reports that considers both individual issue data and data collected from groups of similar defects. The SiameseQAT combines a unique loss function called Quintet Loss that takes the centric of duplicate clusters and their context into account with traditional supervised learning methods for both context and semantics on structured and unstructured characteristics and features based on corpus topic extractions. The method uses over 500,000 bug complaints from the popular open-source software repositories Eclipse, NetBeans, and Open Offices. An evaluation of duplicate retrieval and classification was conducted, revealing results that were much better than prior efforts on both fronts [11].

The words "non-functional requirements," "architectural significant requirements," and "quality attributes" are often used to describe issues linked to quality. These characteristics as a whole influence the software system's non-behavioral issues, such as its dependability, usability, security, and maintainability. These systems tend to degrade over time because of prolonged maintenance efforts, leading to system-wide failures that manifest as quality-related problems. Problems with the system's quality may compromise its reliability and, more importantly, its capacity to perform its essential functions [12]. Manually checking for these high-impact quality-related errors may become a costly and time-consuming endeavor for engineers. Bug reports from larger or more complex programs, like Eclipse, typically have this problem. To address this issue, quality-based classifiers that can automatically identify these newly emerging quality issues from bug report summaries' textual descriptions are developed. The Random Forest ensemble learning technique, in conjunction with a weighted mix of semantic, lexical, and shallow characteristics, is used. Finally, using the Derby project as an example, examine the actual use of classifiers for mapping and visualizing quality-related issues into the code bases. In a nutshell, this work exemplifies early efforts and awareness toward enhancing the foundational management of issue-tracking systems and stakeholders' needs in open-source communities [13].

Developers may get valuable debugging information from the logs included in bug reports. During debugging, developers must analyze the user's bug report and any given logs to reconstruct the sequence of events that resulted in the error. Logs submitted by users are a useful source of information for developers since they demonstrate the issues that users face. However, engineers may have trouble accurately diagnosing the fault if the logs they get are unreliable or missing crucial information. In this research, we perform an empirical study to examine the difficulties and advantages of developer usage of user-provided logs. Focus on the log fragments and exception stack traces that are included in reported bugs [14]. Many reported bugs, however, do not share any characteristics with the classes that have been corrected. Thirdly, manual research shows that the logs don't always include all the details about how the system was run. Many logs just reveal the failure's symptoms (such as an exception) and provide no immediate clue as to the underlying cause. The call graph analysis reveals that in 28% of the sampled bug reports, the corrected class is accessible from the logged class while being hidden from view in the logs that accompany the complaints. In addition, as the system matures, certain logging lines may be deleted from the source code, which might make it more difficult to analyze the logs. In a nutshell, results point to potential future research approaches that might aid practitioners by making it easier for them to attach or analyze logs in problem reports [15].

PROPOSED SYSTEM

The objective is to create a model of bug report similarity that can determine whether a freshly filed report is a copy of an existing report based on its similarities in surface characteristics and textual semantics. Due to the high proportion of duplicate reports (25.9% in the sample), automating the bug triage processes would provide

developers with more time to concentrate on fixing actual bugs and making their products more reliable. The formal model forms the basis of defect report screening infrastructure. Each issue reported in a bug tracker is parsed for a subset of information. The model utilizes the values of these characteristics to foretell whether a newly received bug report will end up being duplicated. To save unnecessary triage time and expenses, duplicate problems are not submitted to developers. The classifier is based on a linear regression on bug report attributes. Incorporating textual semantic data while still performing real-time identifications is made possible by the use of linear regression, which also offers the benefits of (1) having off-the-shelf software supports, lowering the barrier to entry for using systems, (2) supporting rapid classification, and (3) easy components examinations, allowing for a qualitative analysis of the features in the models using cost aware algorithm. To create binary classifiers, we need to provide the features and the output values cutoff that separates duplicates from non-duplicates status since linear regressions yield continuous output value as functions of continuously valued features. Not only base features on the most recently filed bug report but also on a database of all submitted bug reports. The method relies on the premise that these characteristics are adequate to identify duplicates from unique records. Argue that it is possible to distinguish duplicate bug reports from original ones by comparing them to a historical database. Figure 1 shows the system architecture of the proposed system.

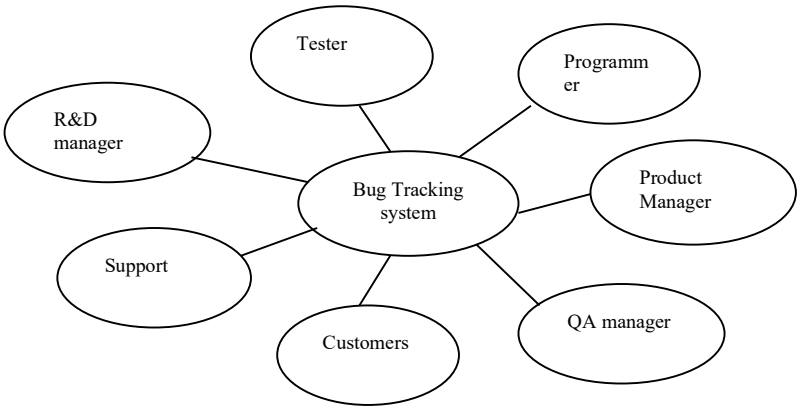

FIGURE 1. Proposed hybrid ensemble classifier for Covid-19 disease

There is too much work involved in updating the model's context data every time a new defect is reported. By allowing the coefficients to be recalculated using previously collected bug data, the linear regression approach expedites the handling of incoming complaints. Only feature extractions, multiplications, and a test against a cutoff are needed to generate a new report. The original historic corpus becomes less important to forecast future duplication, however, as more reports are received. As such, suggest a method in which the fundamental model is regularly (e.g., annually) regenerated, with the coefficient and cutoffs being recalculated based on a newly collected set of reports.

Now, explain how the classifier model's key attributes were derived. Most duplicate bug reports include very similar wording in their free-form textual descriptions and names. To begin, we will establish a textual distance measure appropriate for usage with headings and summaries. This statistic is a crucial part of the system for detecting and eliminating duplicates. When determining how closely two texts are the same, use a "bag of words" technique. Positional information is not stored, and each text is instead seen as a collection of words and their relative frequencies. Some potentially useful semantic information is lost because orders are not retained. The advantage is that the representation's size increases at most linearly with the description's length. This is preferable for a real-time system since it lessens the burden on its processors. Split the titles and body text of bug reports into two distinct corpora. Postulate that titles and descriptions should be considered separately when sorting duplicates. Found that duplicate bug reports are more likely to have identical titles than body text since people tend to be briefer when writing titles. Combining titles and descriptions into a single corpus would result in a loss of information.

Some support for these phenomena may be found in existing literature; for example, a test in which the title is given twice as much importance leads to enhanced performance. Tokenizing the text into individual words and then eliminating the stems allows us to analyze the text more effectively. The MontyLingua tool, along with some simple scripting, is used to extract raw defect reports' descriptions and titles into tokenized, stemmed word lists.

Tokenization eliminates all non-alphanumeric characters, including punctuation, capitalization, and digits. In this research, the widely known Porter stemming method, which allows for more accurate comparisons between individual bug reports by constructing a more standardized corpus, is employed. Next, check each sequence against a dictionary of frequently used terms. Using a stop list, you can get rid of words like "a" and "and" that are in the text but don't really add anything to the content. If these phrases remained, it would give the impression that more detailed defect reports were really the same. The ReqSimile tool's open-source-related stop list of 430 words was utilized. Finally, do not consider submissions-related information, such as the reporter's browser versions, to be part of the description text when defects reports are submitted using a web form. In bug databases, such details are often included next to the description; however, they only contain textual details supplied by the reporters themselves.

RESULTS AND DISCUSSIONS

Build a linear model using features based on textual similarity and clustering findings. Similarity between titles and descriptions is handled independently. Compare the incoming bug report under review to all of the reports in the database, looking for the one with the most similarities in both title and description. Intuitively, if the numbers are small, it's safe to assume that the incoming bug reports are not duplicates of any previously reported issue. In addition, utilize the clusters to build a feature that tracks whether a given report belongs to a certain cluster. The clustering method is more likely to miss a duplicate report if it leaves it alone as a singleton. We expect that graph clustering will help us identify patterns when a single defect has several copies. In the end, round up the model using surface characteristics extracted directly from the defect report. The severity and number of patches or screenshots are all characteristics that may be found here. Neither semantically rich nor predictive, these characteristics pale in comparison to textual similarity. To represent categorical characteristics like the applicable operating system, a one-hot encoding was used.

Research is grounded on over 29,000 Mozilla bug reports. The time frame covered by the reports is from February 2005 to October 2005, a total of eight months. In addition to browsers, mail clients, calendar apps, and bug trackers are also part of the Mozilla project. Data collection contains reports from the same issue-tracking system used by all Mozilla projects. To broaden the applicability of findings, we decided to include all the initiatives. Ever since its inception in 2002, Mozilla has been continuously improved using a cost-aware algorithm. It's possible that early project bug reports aren't reflective of the "typical" or "steady state" for a project's bug reports. Choose a window within which enough reported bugs were fixed while still avoiding potentially problematic conditions at startup. The percentage of unresolved defect complaints in the data set was 17%. Don't utilize reports that don't include developer resolutions since they are considered the gold standard for performance evaluations. At last, only look at duplicates in this dataset if the original problem is present. Figure 2 shows the duplicate bug reports, and Figure 3 shows the true and false positive reports.

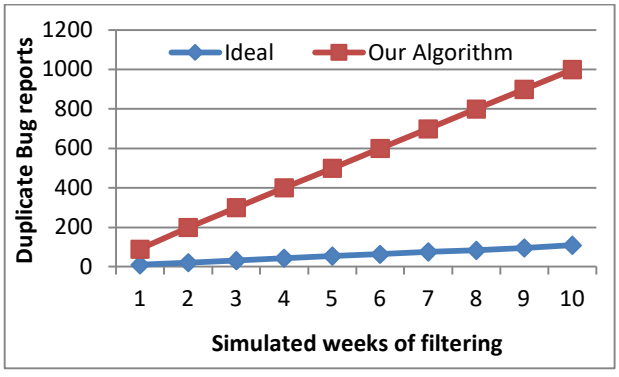

FIGURE 2. Performance of bug tracking system

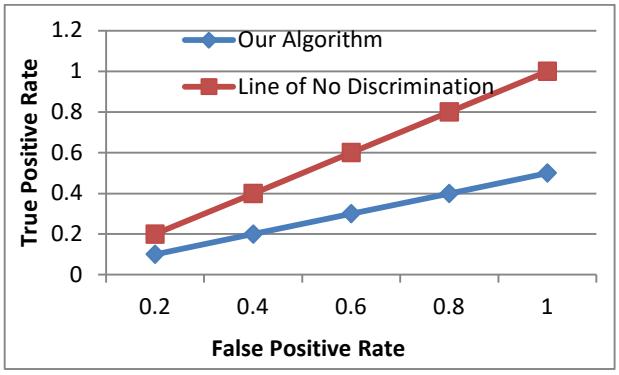

FIGURE 3. Performance of bug tracking system

Four empirical tests were performed: In the initial trial, no evidence using a "rare" phrase together meant you were a duplicate. There was no correlation between the presence of "rare" terms in duplicate bug reports and the presence of such words in non-duplicate bug reports in the dataset. The algorithm's usage of a particular sort of textual similarity measure is inspired by this discovery. The second study replicated and directly compared those results. Each method is given a collection of previously reported bugs and a known duplicate report and then asked to identify potential original reports for the duplicate. The algorithm is effective if and only if the true original is included. Compared to the advanced, the results are not any worse. The online identification of duplicates was the focus of the third and principal investigation. Evaluated the duplication classifier's potential as a real-time filter. Used the first 50 reports to teach and then test the defect reporting algorithm. The withheld bug reports were tested in order, and their duplication was projected as the testing progressed. The bug-tracking dataset is shown in Table 1.

TABLE 1: Result Description

Product	No of Bugs	No of Components
Eclipse Platform	24775	22
Eclipse JDT	10814	6
Mozilla core	74292	137
Mozilla Firefox	69879	47

Calculated the projected savings and costs of such a filter in addition to measuring the time it takes to process incoming defect reports. Calculated the cost and benefit by tallying the numbers of false positives and true negatives, or actual flaws and duplicates, respectively. The characteristics that the model relied on were then subjected to a leave-one-out analysis and a principal components analysis. These evaluations consider the characteristics chosen and their potential for overlap in terms of their predictive power and degree of similarity.

CONCLUSIONS

To prevent developers from wasting time, suggest a system that may automatically sort incoming problem reports into distinct categories. This method makes use of graph clustering in addition to textual semantics and surface information to make duplication predictions. An actual evaluation of the method was conducted using a dataset of 29,000 bug reports from the Mozilla projects, which is much bigger than the datasets typically described in the literature. Simulating its usage as a filter in a real-time bug-reporting environment demonstrates that inverse document frequency is not helpful for this objective. The solution can eliminate 1% of duplicate bug reports, saving money on development. Although it takes just 20 seconds for every receiving bug report to generate a categorization, it nonetheless ensures that at least one report for each actual problem reaches developers. Therefore, the method's system might be realistically deployed in a production setting with little extra work and potentially substantial benefit. Cost-aware algorithm-based bug tracking systems are evaluated based on how well they manage costs, optimize resources, efficiently resolve bugs, and provide high-quality software overall. Prioritize bugs based on security concerns. Improve the cost-aware algorithm to find and rank security flaws, making sure that serious security issues are fixed quickly to reduce risks.

REFERENCES

[1]. M.K. Gopal, M. Govindaraj, P. Chandra, P. Shetty, and S. Raj, 2022, "Bugtrac–a new improved bug tracking system," *In IEEE Delhi Section Conference*, pp. 1-7.

[2]. M. Haering, C. Stanik, and W. Maalej, 2021, "Automatically matching bug reports with related app reviews," *In IEEE/ACM 43rd International Conference on Software Engineering*, pp. 970-981.

[3]. W. Aljedaani, M.W. Mkaouer, S. Ludi, A. Ouni, and I. Jenhani, 2022, "On the identification of accessibility bug reports in open-source systems," *In Proceedings of the 19th International Web for all conference*, pp. 1-11.

[4]. R. Malhotra, A. Dabas, AS. Hariharasudhan, and M. Pant, 2021, "A study on machine learning applied to software bug priority prediction," *In 11th International Conference on Cloud Computing, Data Science and Engineering*, pp. 965-970.

[5]. M. Soliman, M. Galster, and P. Avgeriou, 2021, "An exploratory study on architectural knowledge in issue tracking systems," *In European Conference on Software Architecture. Cham: Springer International Publishing*, pp. 117-133.

[6]. J. Polpinij, 2021, "A method of non-bug report identification from bug report repository," *Artificial Life and Robotics*, pp. 318-328.

[7]. T.M. Rocha, and A.L. Carvalho, 2021, "SiameseQAT: A semantic context-based duplicate bug report detection using replicated cluster information," *IEEE Access*, **9**, pp. 44610-44630.

[8]. R. Krasniqi, and A. Agrawal, 2021, "Analyzing and detecting emerging quality-related concerns across oss defect report summaries," *In IEEE International Conference on Software Analysis, Evolution and Reengineering*, pp. 12-23.

[9]. A.R. Chen, T.H. Chen, and S. Wang, 2021, "Demystifying the challenges and benefits of analyzing user-reported logs in bug reports," *Empirical Software Engineering*, pp. 1-30.

[10]. G.Rosa, L. Pascarella, S. Scalabrino, R. Tufano, G. Bavota, M. Lanza, and R. Oliveto, 2021, "Evaluating szz implementations through a developer-informed oracle," *In IEEE/ACM 43rd International Conference on Software Engineering*, pp. 436-447.

[11]. H. Liu, Y. Yu, S. Li, M. Geng, X. Mao, and X. Liao, 2021, "How to cherry pick the bug report for better summarization?" *Empirical Software Engineering*, **26**, pp. 1-36.

[12]. T. Zhang, D. Han, V. Vinayakarao, I.C. Irsan, B. Xu, F. Thung, D. Lo, L. Jiang, 2023, "Duplicate bug report detection: How far are we?" *ACM Transactions on Software Engineering and Methodology*, **32(4)**, pp. 1-32.

[13]. X. Xie, Y. Su, S. Chen, L. Chen, J. Xuan, and B. Xu, 2021, "MULA: A just-in-time multi-labeling system for issue reports," *IEEE Transactions on Reliability*, **71(1)**, pp. 250-263.

[14]. M.M. Imran, A. Ciborowska, and K. Damevski, 2021, "Automatically selecting follow-up questions for deficient bug reports," In IEEE/ACM 18th International Conference on Mining Software Repositories, pp. 167-178.

[15]. F.A. Bhuiyan, M.B. Sharif, A. Rahman, 2021, "Security bug report usage for software vulnerability research: a systematic mapping study," *IEEE Access*, **9**, pp. 28471-28495.