The Computer Journal Advance Access published January 28, 2016

doi:10.1093/comjnl/bxv114

A Literature Review of Research in Bug Resolution: Tasks, Challenges and Future Directions

Tao Zhang[1], He Jiang[2], Xiapu Luo[1]*, and Alvin T.S. Chan[3]

[1]*Department of Computing, The Hong Kong Polytechnic University, Hong Kong, China*
[2]*School of Software, Dalian University of Technology, Dalian, China*
[3]*Singapore Institute of Technology, Singapore, Singapore*
**Corresponding author: csxluo@comp.polyu.edu.hk*

Due to the increasing scale and complexity of software products, software maintenance especially on bug resolution has become a challenging task. Generally in large-scale software programs, developers depend on software artifacts (e.g., bug report, source code and change history) in bug repositories to complete the bug resolution task. However, a mountain of submitted bug reports every day increase the developers' workload. Therefore, 'How to effectively resolve software defects by utilizing software artifacts?' becomes a research hotspot in software maintenance. Considerable studies have been done on bug resolution by using multi-techniques, which cover data mining, machine learning and natural language processing. In this paper, we present a literature survey on tasks, challenges and future directions of bug resolution in software maintenance process. Our investigation concerns the most important phases in bug resolution, including bug understanding, bug triage and bug fixing. Moreover, we present the advantages and disadvantages of each study. Finally, based on the investigation and comparison results, we propose the future research directions of bug resolution.

Keywords: bug resolution; bug report; software maintenance; bug understanding; bug triage; bug fixing

Received 7 June 2015; revised 11 November 2015
Handling editor: Rob Hierons

1. INTRODUCTION

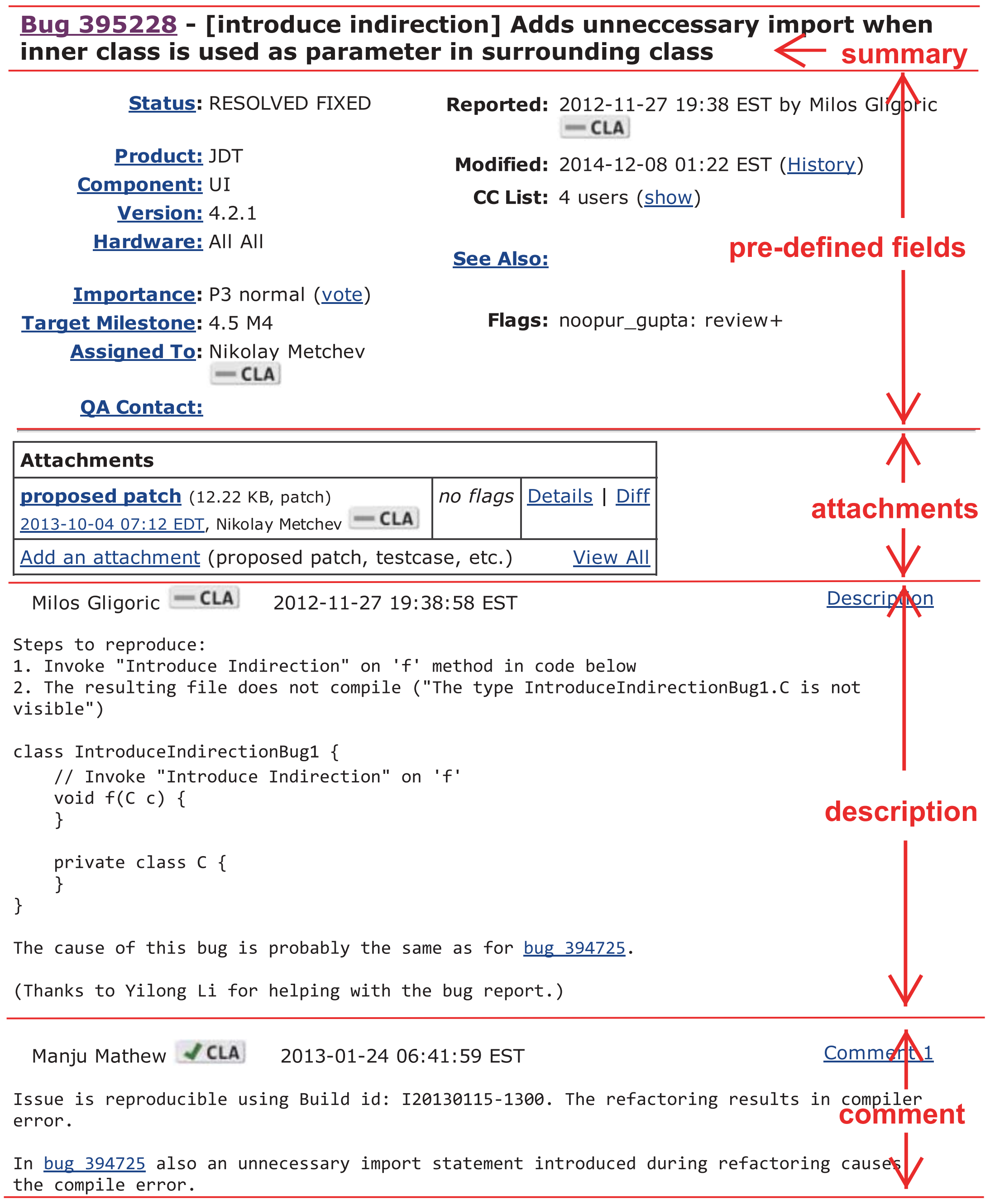
1.1. Background

The ultimate aim of software maintenance is to not only improve the performance but also fix defects and enhance attributes of the software, which lead to better quality software. Software bugs usually appear during software development process, unfound bugs can lead to the loss of billions of dollars [1]. In recent years, software maintenance has become more challenging due to the increasing number of bugs in large-scale and complex software programs. The previous study [2] showed that more than 90% of software development cost is spent on maintenance and evolution activities. To coordinate and avoid overlapping efforts, in large-scale open source software projects, project teams usually utilize the bug tracking systems such as Bugzilla to keep track of reported software bugs. A core component is the bug repository that stores the software artifacts such as bug reports, source code and change history produced by users and developers. The developers in project teams depend on them to manage and fix the given bugs.

In the software maintenance process for large-scale software programs, software artifacts, especially for bug reports, become an important medium to help developers resolve bugs. Specifically, a user or developer can report the software bug in a fixed format (i.e., bug report) and upload it to a bug tracking system such as Bugzilla[1] and FogBugz[2]. Then, a senior developer is assigned to fix the reported bug according the information shown in this submitted report. Figure 1 shows an Eclipse bug report-*Bug 395228* which is a representative sample that contains all basic elements, such as pre-defined fields, freeform text and an attachment.

The pre-defined fields on the bug report provide a variety of descriptive metadata, such as status (e.g., resolved fixed that

[1] https://www.bugzilla.org/.
[2] http://www.fogcreek.com/fogbugz/.



Bug 395228 - **[introduce indirection] Adds unneccessary import when inner class is used as parameter in surrounding class**

Status:	RESOLVED FIXED	**Reported:**	2012-11-27 19:38 EST by Milos Gligoric CLA
Product:	JDT	**Modified:**	2014-12-08 01:22 EST (History)
Component:	UI	**CC List:**	4 users (show)
Version:	4.2.1		
Hardware:	All All	**See Also:**	
Importance:	P3 normal (vote)		
Target Milestone:	4.5 M4	**Flags:**	noopur_gupta: review+
Assigned To:	Nikolay Metchev CLA		
QA Contact:			

Attachments		
proposed patch (12.22 KB, patch) 2013-10-04 07:12 EDT, Nikolay Metchev CLA	*no flags*	Details | Diff
Add an attachment (proposed patch, testcase, etc.)		View All

Milos Gligoric CLA 2012-11-27 19:38:58 EST Description

```
Steps to reproduce:
1. Invoke "Introduce Indirection" on 'f' method in code below
2. The resulting file does not compile ("The type IntroduceIndirectionBug1.C is not
visible")

class IntroduceIndirectionBug1 {
    // Invoke "Introduce Indirection" on 'f'
    void f(C c) {
    }

    private class C {
    }
}

The cause of this bug is probably the same as for bug 394725.

(Thanks to Yilong Li for helping with the bug report.)
```

Manju Mathew ✓CLA 2013-01-24 06:41:59 EST Comment 1

```
Issue is reproducible using Build id: I20130115-1300. The refactoring results in compiler
error.

In bug 394725 also an unnecessary import statement introduced during refactoring causes
the compile error.
```

FIGURE 1. An example of Eclipse bug report 395228.

stands for the bug report was resolved), importance (e.g., ‘P3 normal’ consists of the priority ‘P3’ and the severity ‘normal’), component (e.g., UI) and product (e.g., JDT). These metadata contain the basic features of the bug report. The freeform text consists of a summary, a description and comments. The summary and description present detailed information of the reported bug, and the comments are posted by developers who participate in or have interests in the project and discuss how to fix the bug. Generally, developers can attach non-textural information to the bug report, such as patches and test cases.

For each bug report, the status in the pre-defined fields indicates its life-cycle. Figure 2 shows the general life-cycle of bug reports in Bugzilla. When a new bug report is submitted by a reporter, the initial state is changed from ‘Unconfirmed’ to ‘New’ after the bug report is verified by a triager who is usually a senior developer or project manager. The triager is also

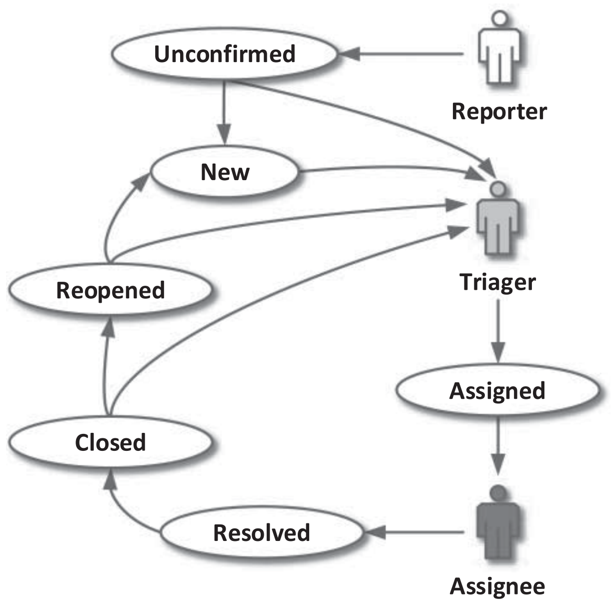


FIGURE 2. The life cycle of bug resolution process in Bugzilla.

responsible to assign the bug report to an appropriate developer to fix it. After that, a developer is nominated as the assignee and the state of the bug report is changed to 'Assigned'. If the assignee completes the bug-fixing task, the state is changed to 'Resolved'; otherwise, the triager will attempt to identify and assign other developers to resolve the bug. Once the triager verifies that the bug is fixed successfully, she or he ends the task and the state becomes 'Closed'. Afterward, if a developer finds that the bug is not fixed in its entirety, this bug can be reopened by the triager. The bug-fixing task is re-executed in a step-wise manner through a cycle-regulated process as described above.

In the life-cycle of bug resolution process, a new submitted bug report needs to undergo three phases, including bug understanding, bug triage and bug fixing so that the reported bug can be fixed. Figure 3 shows the three phases and related tasks in all life-cycle of bug report. We detail them as follows:

(1) *Bug understanding*: In the process of changing the status from 'Unconfirmed' to 'New', triagers need to fully understand the contents of given bug reports so that they can summarize the important contents, filter the duplicates and predict the features (e.g., priority, severity and status for reopened/blocking) of reported bugs. The major challenge results from the huge amount of bug reports submitted every day. A mountain of work can lengthen the fixing time and affect the quality of task execution. Existing studies [3–35] aim at developing automatic approaches to perform these tasks.

(2) *Bug triage*: As a common process, triagers are responsible for arranging the right developers to fix the given bugs, and then mark the status of corresponding bug reports to 'Assigned'. Without a good understanding of the bug reports, triagers may assign improper developers to execute the task of bug fixing. It leads to the bug re-assignment(s). In other words, the related bugs need to be reassigned to other developers. Unfortunately, Jeong *et al.* [36] showed that the more the number of reassignments is, the lower the success probability of bug fixing is. Except for reassignments, processing a multitude of bug reports places a heavy burden on triagers. To resolve this problem, several automatic bug triage approaches [29, 30, 36–53] have been proposed to recommend the best developers for fixing the given bugs.

(3) *Bug fixing*: The assigned developers, usually called as assignees, are responsible for fixing the reported bugs. For a given bug, the assignee needs to find the source code files where the bug is, and develop or update patch code as a part of the bug-fixing process. Manual bug localization and patch generation may render the bug fixing process for fixing intractable due to the ever-increasing bugs. Thus, researchers have developed automatic tools to locate the given bugs [54–66] and generate the patches [67–71].

In Fig. 3, we present six tasks in three phases. Existing studies aim to perform them automatically. After utilizing natural language processing techniques, including tokenization, stemming and stop words removal, to pre-process the corresponding software artifacts (e.g., bug reports, source code) in software repositories, they adopted multiple techniques such as data mining (DM) and machine-learning techniques to execute the tasks. In this paper, we survey the previous papers covering six tasks in three phases of bug resolution, and compare the advantages and disadvantages of each proposed approach. Moreover, we present the future directions of this research field.

1.2. Survey process

We search the renowned journals and conference proceedings to find the corresponding papers concerning the above-mentioned three phases in bug resolution since 2004. The selected journals and conference proceedings mainly come from Software Engineering (SE) field. To keep the wide coverage, we also include some papers published in other fields such as DM field, because these papers have strong connection with the topics studies in this paper. Table 1 introduces the sources of papers covering bug resolution. Note that 74 journal and conference papers (five papers were published in the journal and conference proceedings of other fields) were reviewed, we checked each paper to guarantee the correctness and relevance.

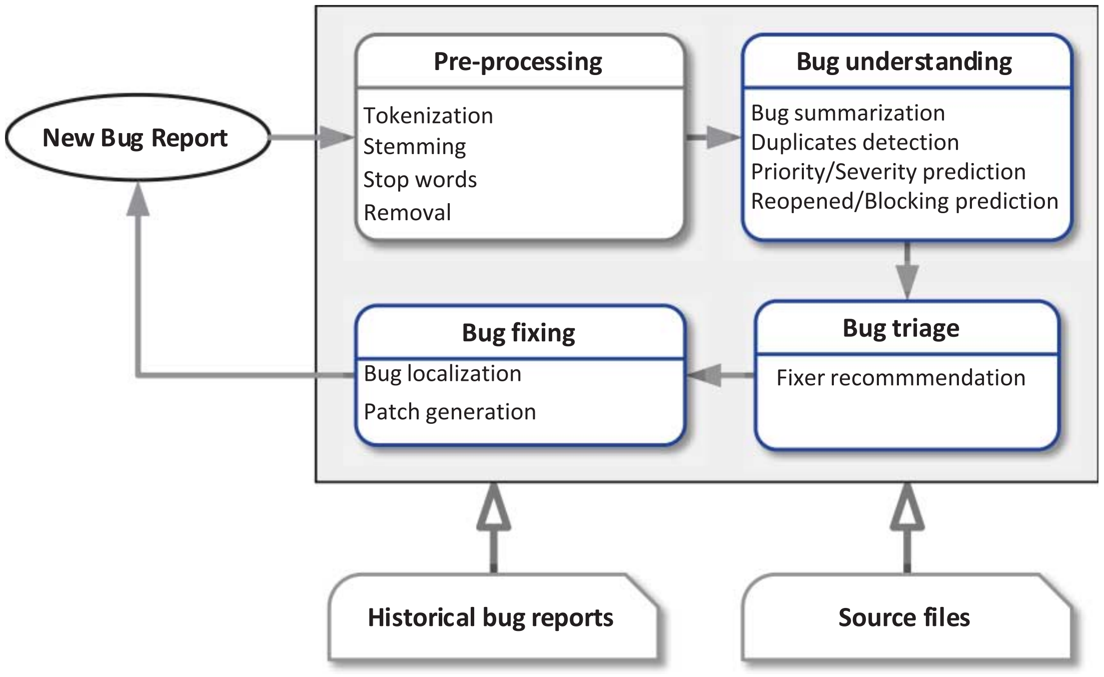


FIGURE 3. Framework for bug resolution.

1.3. Inclusion and exclusion criteria

In the process of paper selection, the articles which satisfy the following options were included in our survey:

(1) Papers must describe the methodology and experiments. For example, the researchers should describe how to utilize the proposed new algorithms to resolve the given bugs.
(2) Papers must focus on the six tasks (i.e., bug summarization, duplicate detection, feature prediction, bug triage, bug localization and patch generation) in the three phases of bug resolution, including bug understanding, bug triage and bug fixing.
(3) Papers were published in peer-reviewed journals and conference proceedings, because they are more representative.

In addition, we exclude the papers which concern the following problems:

(1) Papers do not show how to conduct the experiment and the corresponding evaluation results.
(2) Papers are the duplicates of same study.
(3) Papers are not relevant with the survey topics in our work even though they may concern the software artifacts such as bug reports or source code. For example, papers discussing the classification of bug reports are out of scope of our survey.

1.4. Analysis and classification method

We analyzed all papers appearing in Table 1 that we selected according to their categories. In this survey paper, we classify each selected paper into a specific category for each task based on the different algorithms presented in the papers. For example, for the task of bug triage, some studies like [37, 38] utilized machine-learning algorithms to recommend the appropriate bug fixers while [30, 50, 51, 53] used topic model to complete the same task. On this occasion, we classify the former into the class 'Machine-learning-based recommender', and categorize the latter into the class 'Topic model-based recommender'.

1.5. Contribution: different points against prior surveys

This work presents the contributions to three phases of bug resolution via software artifacts. Only a few previous surveys [72, 73] are relevant to our work. In [72], Strate and Laplante focus on bug reports analysis, including duplicates detection and bug triage. Similar to [72, 73] mainly concerns the analysis of bug reports, and discusses the SE tasks such as duplicates detection and bug localization.

We have the following new contributions, which differentiate our work from prior survey papers.

(1) Our work reviews papers in the three phases of bug resolution, which are much more than papers on the analysis of bug reports.

TABLE 1. Sources of papers reviewed.

SE Journal	Acronym	No. of papers
IEEE Transactions on Software Engineering	TSE	4
ACM Transactions on Software Engineering and Methodology	TOSEM	1
Information and Software Technology	IST	2
Journal of System and Software	JSS	1
Automated Software Engineering	ASE	1
Empirical Software Engineering	EMSE	1
Total		**10**
SE Conference Proceedings		
International Conference on Software Engineering	ICSE	16
International Symposium on the Foundations of Software Engineering	FSE	3
International Conference on Automated Software Engineering	ASE	4
Working Conference on Reverse Engineering	WCRE	6
European Conference on Software Maintenance and Reengineering	CSMR	4
International Conference on Software Analysis, Evolution, and Reengineering	SANER	1
Working Conference on Mining Software Repositories	MSR	8
International Conference on Software Maintenance and Evolution	ICSM(E)	6
International Conference on Program Comprehension	ICPC	1
International Symposium on Empirical Software Engineering and Measurement	ESEM	1
International Conference on Software Engineering and Knowledge Engineering	SEKE	2
International Conference on Predictive Models in Software Engineering	PROMISE	1
Asia-Pacific Software Engineering Conference	APSEC	4
International Conference on Software Engineering Advances	ICSEA	1
International Computer Software and Applications Conference	COMPSAC	2
International Conference on Software Testing, Verification and Validation	ICST	1
Total		**59**
Others		
Journal of Computer Science and Technology	JCST	1
International Conference on Dependable Systems and Networks	DSN	1
International Conference on Advanced Data mining and Applications	ADMA	1
International Conference on Intelligent Systems Design and Applications	ISDA	1
International Conference on Machine Learning and Applications	ICMLA	1
Total		**5**
Total Survey Papers		**74**

(2) The research objects of reviewed papers not only concern the bug reports but also include other software artifacts such as source code and change history.
(3) The surveyed range of our work is more than other prior surveys. In detail, we include the related papers published in 2015, but [72, 73] do not cover the up-to-date research articles.

Roadmap. We organize this article as follows, in Sections 2–4, we categorize the previous studies as the different phases of bug resolution, and analyze how previous studies realize these approaches by utilizing the contents of bug reports and related data resources (e.g., source code file). In Section 5, we describe the techniques that were utilized in the previous studies, and discuss the elements which can impact the performance of tasks in the bug resolution process. We introduce the future research directions in Section 6. In Section 7, we conclude this survey paper.

2. BUG UNDERSTANDING

As the description in Section 1, the triager needs to understand the content of each new submitted bug report rapidly so that he/she can verify whether it is a duplicate of existing bug reports. Moreover, he/she needs to mark the features such as priority and severity levels or verify whether the reporter's annotation is correct or wrong. To reduce the triagers' workload when they face to a lot of new bug reports, existing studies proposed a series of automatic approaches to implement bug report summarization, duplicates detection and feature prediction. In this section, we introduce these works and summarize their advantages and disadvantages.

2.1. Survey on bug report summarization approaches

A vast number of bug reports often contain excessive description and comments, which may become a burden to the developers. Automatic summarization of bug reports is one way to help developers reduce the size of bug reports. As a frequently used method, extraction approach selects a subset of existing sentences to produce the summary so that it can be utilized to produce the summary of bug reports. For bug report summarization, the major challenge is how to select the sentences from duplicate bug reports to generate the summary of given bug reports. To resolve this problem, supervised learning and unsupervised learning approaches can be adopted to determine which sentences are appropriate. We introduce how the previous studies implement two different approaches in the following subsections.

2.1.1. Supervised learning approaches

Supervised learning is a type of machine-learning algorithm that builds a prediction model by training the labeled data to execute the prediction task [74]. It is necessary to label the data before performing a training process. Kupiec *et al.* [75] first proposed the use of supervised machine learning for document summarization. They think that the supervised learning approach can freely use and combine any desired features. Supervised learning approaches rely on the labeled summary corpus for training to predict which sentences belong to a part of the summary of a new bug report. Rastkar *et al.* [3] firstly invited the human annotators to create summaries for 36 bug reports, and then they applied three supervised classifiers such as Email Classifier (EC), Email & Meeting Classifier (EMC) and Bug Report Classifier (BRC) to verify whether a sentence is a part of the extractive summary. By measuring the effectiveness of these classifiers, they found that BRC out-performs the other two classifiers by 20% in precision when producing summaries of bug reports.

In an extended version (i.e., [4]) of [3], Rastkar *et al.* demonstrated whether the generated bug report summaries can help users perform the detection of duplicate bug reports better. As a result, they found that the summaries helped the participants save time with no evidence that the accuracy of duplicates detection has been fallen off.

Note that supervised learning-based summarization approaches still need the manual effort on building corpus and gold summarizes, therefore the execution cost is increased obviously.

2.1.2. Unsupervised learning approaches

Different from supervised learning approaches which require large supervised or labeled data, the training data for unsupervised learning approaches are not labeled (i.e., merely the inputs) [76]. Mani and his colleagues [5] applied four unsupervised approaches, including Centroid, Maximum Marginal Relevance (MMR), Grasshopper and Diverse Rank (DivRank), to summarize the given bug reports. At first, they designed a noise reducer to filter out redundant sentences from historical bug reports and then executed four unsupervised algorithms to generate the corresponding summaries. Unsupervised summarization methods choose sentences that are central to the input bug report. This centrality can be measured in various ways. For example, centroid is a simple technique to achieve this goal. In this algorithm, each sentence of bug report is represented as a weighed vector of $tf - idf$ [77]. For each sentence, the algorithm defines *Centroid Value*. It is calculated as the sum of the term weights in the centroid sentence, which is a pseudo-sentence whose vector has a weight equal to the average of all the sentence vectors in the report. When the centroid values of sentences have been calculated, the summary is formed by selecting sentences in the decreasing order of their centroid values. Mani *et al.* compared the efficiency of the proposed unsupervised methods with supervised approaches proposed by Rastkar *et al.* [3] The evaluation results showed that MMR, DivRank and Grasshopper algorithms performed on par with the best of the supervised approach but saved on the running cost because it avoids manual annotation.

Lotufo *et al.* [6] developed an unsupervised summarization approach to generate the summaries of given bug reports without need for configuration nor of manual annotation. This approach is based on a hypothetical model which assumes the reader will have to review a lot of sentences within a limited time and focus on the important ones. The authors create three hypotheses on what kinds of sentences a reader would find relevant: sentences that highlight frequently discussed topics, sentences that are evaluated by other sentences, and sentences that focus on the topics in the title and description of the bug reports. By using this hypothetical model, they applied PageRank [78] to calculate the probabilities of sentences being read and compose the summary of the given bug report. The experimental results showed that the proposed approach improved the performance of automatic bug report summarization by comparing with EC developed by Rastkar *et al.* [3].

2.1.3. Comparison of summarization approaches

We provide the detailed comparison between different summarization approaches [3–6] in Table 2. Note that [5, 6] utilized unsupervised approaches to automatically generate summaries of bug reports. By comparing with [3, 4] that adopted supervised classifiers, they achieved satisfactory precision, while avoiding the human-annotation.

TABLE 2. Comparison of automatic summarization approaches.

Works	Mechanism	Method summary	Advantages	Disadvantages
[3, 4]	Supervised classifier	Utilized EC, EMC and BRC to generate the probability of each sentence being part of a summary	Demonstrated that BRC achieved <60% precision that outperformed EC and EMC classifier for 36 bug reports chosen from Eclipse Platform, Gnome, Mozilla and KDE.	Needs to create the human-annotated standard summaries
[5]	Unsupervised summarizer	Utilized centroid, MMR, Grasshopper and DivRank to summarize bug reports	Avoids human-annotation and reduces the noise data	Factors may affect the dependent variables
[6]	Unsupervised summarizer	Modeled a user reading process and used this model to rank sentences for generating summaries	Improved the performance of [3] by up to 12% in terms of precision for a random set of bug reports from the Debian, Launchpad, Mozilla and Chrome projects	Additional cost to test the hypotheses that decided the rank of each candidate sentence

2.2. Survey on duplicates detection approaches

Duplicate bug reports [79] occur when more than one developer submits a similar description for the same bug. Verifying whether a new bug report is a duplicate or not is a major task of triagers before bug assignment. Unfortunately, it is a tedious and time-consuming work for triagers due to potentially large number of submitted bug reports and diversity in reporting style of users. Automatic duplicate detection is necessary in order to avoid the manual process. How to train an effective discriminative model to check whether a newly arrived bug report is a duplicate or non-duplicate of existing bug reports becomes a challenging problem. In recent years, there are a number of research works [7–17] on automatic duplicates detection to identify and locate duplicates. We classify these approaches into two categories, including textual information analysis-based detection and hybrid information analysis-based detection, which are introduced in the following subsections.

2.2.1. *Textual information analysis-based detection*

Textual information analysis-based approaches only utilize the provided textual contents such as summaries, descriptions and comments of bug reports to detect the duplicates. As an early study, Runeson *et al.* [7] applied NLP techniques to process bug reports, then adopted Vector Space Model (VSM) [80] and cosine similarity [81] to measure the textual similarity between a new bug report and historical reports so that they can identify which reports are duplicates of the given bug report. The evaluation showed that about tow-thirds of the duplicates can be found.

Sun *et al.* [10] developed a discriminative model to check whether a new bug report is duplicate or not. In this study, they only considered to use *idf* which is inverse document frequency for getting the term weights and utilized them to get the textual similarity between two bug reports. In detail, the authors denoted the three types of *idf* computed within the three corpora by idf_{sum}, idf_{desc} and idf_{both}, respectively. For three corpora, one corpus is the collection of summaries, one corpus is the collection of descriptions and the other is the collection of the hybrid data sets including both of summaries and descriptions. They calculated the different similarities according to the different types of *idf* and employed them as the features of Support Vector Machine (SVM) so that discriminative model can be developed to detect the duplicates of the given bug report. The evaluation results showed that this duplicates detection method outperformed the approaches proposed in [7].

Different from the previous studies such as [7, 10] which adopted word-level representation of bug reports, Sureka and Jalote [15] proposed a new representation way, namely character-level representation to express the title and description of each bug report. By building the character-level n-gram model, they can capture the important linguistic characteristics (i.e., discriminatory features) of bug reports to execute the duplicates detection. This approach is evaluated on a bug repository consisting of >200 thousand bug reports from Eclipse project. The recall rate for the top-50 results reached up to 33.93% for 1100 randomly selected test cases and 61.94% for 2270 randomly selected test cases.

2.2.2. *Hybrid information analysis-based detection*

In order to further improve the accuracy of duplicates detection, combining other non-textual information is necessary. As a early study, Jalbert and Weimer [8] built a classifier for given bug reports that combines their surface features (e.g., severity and daily load), textual similarity metrics and graph clustering algorithms to identify duplicates. The proposed approach performed up to 1% better than that of the study proposed by Runeson *et al.* [7] and reduced the development cost by filtering out 8% of duplicate bug reports.

Wang and his group [9] improved the accuracy of duplicates detection by combining natural language description and execution information (i.e., execution traces) of reported bugs. In detail, they calculated the natural language-based similarities and the execution information-based similarities between a new bug report and existing bug reports by using VSM and cosine similarity measure; then they combined two similarities to detect duplicate bug report. The experimental results showed that this hybrid approach can detect higher-proportional duplicate bug reports than the method [7] using natural language information alone.

Sun and his group [11] proposed REP, which is a new similarity measure function, which can not only compute the similarity of textual content in summary and description of bug reports but also calculate the similarity of non-textual fields such as product, component and version. $BM25F_{ext}$ was introduced to compute the similarities. By using REP, they can effectively detect the duplicate of bug reports. They validated this technique on three large software bug repositories from Mozilla, Eclipse and OpenOffice, the results showed that it can improved the accuracy over SVM-based duplicates detection.

Tian *et al.* [16] utilized REP [11], which adopted an extension of $BM25F$ to measure the similarity of two bug reports, and adopted the feature, i.e., 'product' of bug report, to help in identifying whether two bug reports are duplicate or not. Furthermore, they defined a new notion of relative similarity that help to decide if the similarity between two bug reports is significant. The experimental results showed that this approach can improve the accuracy of the previous study [8] by ∼160%.

Nguyen and colleagues [12] not only used $BM25F$ to compute the textual similarity between a new bug report and historical bug reports but also calculated the similarity between the new report and the duplicate report groups which could share same topic(s) built by Latent Dirichlet Allocation (LDA) [82]. By combining the textual similarity via $BM25F$ and the topic similarity, Nguyen *et al.* can effectively detect the duplicates of the given bug report. The evaluation results demonstrated that the proposed technique improved the approach in [11] by up to 20% in accuracy.

In [13] and its extended version [14], Alipour *et al.* developed a contextual approach to further improve the accuracy of duplicates detection. Specifically, they built a contextual word collection, including six word lists labeled as efficiency, functionality, maintainability, portability, reliability and usability, then computed the similarity between these word lists and each bug report using $BM25F$. Moreover, they combined the primitive textual and categorical features of bug reports, such as description, component, type and priority. Finally, some well-known machine-learning techniques such as C4.5 [83], K-Nearest Neighbor (KNN), Logistic Regression [84] and Naive Bayes were applied to execute the detection task for duplicate bug reports. Alipour *et al.* demonstrated that the contextual duplicates detection method performed better than the approach provided by Sun *et al.* [11].

Aggarwal *et al.* [17] proposed a method called the software-literature context method, which utilized the lists of contextual words from SE textbooks and project documentation to detect the duplicate bug reports via the similarity measure $BM25F$. The adopted lists are different from [13], these software-literature context word lists reflect the software development processes, and the evolution of project. The experimental results are similar to the results reported by Alipour *et al.*, however, this approach requires far less time and effort. For example, the SE textbook and project documentation features were 1.48% less accurate on Android dataset than labeled LDA features utilized in [13] but these features took only 0.5 h to extract, while the labeled LDA features took 60 h to extract due to annotation.

Note that hybrid information can help to improve the performance of duplicates detection, however, these approaches need to extract further features or build the complex model (e.g., topic model) to implement the task. In the following subsection, we summarize and compare the above mentioned approaches on duplicate bug reports detection.

2.2.3. *Comparison of automatic duplicate detection methods*

We summarize the comparison of different automatic duplicate detection approaches [7–17] in Table 3. Note that multiple information retrieval (IR) and machine-learning techniques are effective ways to detect the duplicate bug reports in these studies. Moreover, different features lead to different detection accuracy. For example, due to additional features such as contextual and categorical information, the proposed approach in [13] performed better than the method in [11].

2.3. Survey on feature prediction approaches

Features (e.g., priority, severity and status for reopened/blocking) of reported bugs can provide clear guide to developers in fixing the given bugs. In detail, the priority helps developers verify which bugs should be given attention first; the severity is a critical feature in deciding how soon the bug needs to be fixed; the status for 'reopened' means that the bugs should be reopened due to existing unsolved problems, and 'blocking' bugs are software bugs which can prevent other bugs from being fixed, both of them can increase the software development cost and the workload of bug fixers. Thus, it is necessary to predict the corresponding features of bug reports for helping to improve the bug-fixing process. We group these features into two difference categories (i.e., priority/severity and reopened/blocking) due to similar characteristics, which are presented as following subsections.

TABLE 3. Comparison of automatic duplicates detection approaches.

Works	Mechanism	Method summary	Advantages	Disadvantages
[7]	Textual information analysis-based detection	Adopted VSM and cosine measure to compute the textual similarity between a new bug report and historical bug reports so that duplicates were found.	Reducing the effort to identify duplicate reports with 40% at Sony Ericsson Mobile Communications.	Relatively lower accuracy (42% for a top list size of 15, the maximum recall rate can achieve 66.7%).
[10]	Textual information analysis-based detection	Built a discriminative model via SVM to verify whether a given bug report is the duplicate or non-duplicate report.	Achieved 17–31%, 22–26% and 35–43% relative improvement over [7–9] in OpenOffice, Firefox and Eclipse, respectively.	Needs more features.
[15]	Textual information analysis-based detection	Built a character-level n-gram model to detect the duplicates	Run at the large-scale data sets (200 thousand bug reports from Eclipse project)	Relatively lower recall rate (33.92% for 1100 random cases and 61.94% for 2270 random cases).
[8]	Textual information and feature analysis-based detection	Built a classifier for that combines surface features, textual similarity metrics, and graph clustering algorithms to identify duplicates.	Performed better than [7] by up to 1% accuracy and reduce the development cost for a dataset of 29,000 bug reports from the Mozilla project.	Performance improvement (∼1%) is not obvious and needs more features.
[9]	Textual information and execution information analysis-based detection	Used VSM and cosine measure to compute the textual similarity and execution information similarity between a new bug report and historical bug reports, then combined them as a hybrid algorithm to detect the duplicates.	Detected 67–93% of duplicate bug reports in the Firefox bug repository, which is much better than 43–72% using natural language alone.	Increased the complexity of the algorithm.
[11]	Textual information and feature analysis-based detection	Utilized $BM25F_{ext}$ to compute the textual similarity and feature (i.e., product and component) similarity between a new bug report and historical bug reports to detect the duplicates.	Achieved 10–27% relative improvement in recall rate than SVM-based detection method, and improved the recall rate of [15] by up to 37–71% on three bug repositories from Mozilla, Eclipse and OpenOffice.	Needs more features.
[16]	Textual information and feature analysis-based detection	Utilized REP and the feature ‘product’ to identify the duplicates.	Improve the accuracy of [8] by 160% for the Mozilla project.	Needs more feature.
[12]	Textual information and topic analysis-based detection	Used $BM25F$ to compute the textual similarity between a new bug report and historical bug reports, and also calculated the similarity between the new report and the duplicate groups sharing the same topic(s); then combined two similarities to verify whether the new bug report is duplicate or non-duplicate.	Improved over [11] with higher accuracy from 4 to 6.5% for OpenOffice and 5–7% for Mozilla.	Needs to adjust the number of topics while running the algorithm in different data sets.

continued.

TABLE 3. continued.

Works	Mechanism	Method summary	Advantages	Disadvantages
[13, 14]	Textual information and contextual analysis-based detection	Introduced *BM*25*F* to calculate the contextual similarity between the word lists and a given bug report, then combined the textual and categorical features as the input of multiple machine-learning algorithms (e.g., KNN) so that the duplicates were detected.	Improved prediction accuracy by 11.55% over [11] for Android, Eclipse, Mozilla and OpenOffice projects.	Needs the contextual word lists from bug reports.
[17]	Textual information and domain knowledge analysis-based detection	Utilized the lists of contextual words from SE textbooks and project documentation to detect the duplicate bug reports via *BM*25*F*.	Required only 0.5 h to extract the features while the approach [13] spent 60 h to extract them at Android, Eclipse, Mozilla and OpenOffice projects.	Needs the domain knowledge of SE field.

2.3.1. Priority/severity prediction approaches

For each bug report, in terms of precedence, generally there are five priority levels that can be denoted as *P*1, *P*2, *P*3, *P*4 and *P*5; *P*1 being the highest priority, while *P*5 represents the lowest priority. The severity varies from *trivial*, *minor*, *normal*, *major*, *critical* to *blocker*, which indicate the increasing severity of the given bugs. Bug triagers need to understand the information provided by reporters in the submitted bug reports and decide the appropriate factors. Even if clear guidelines [85] exist on how to assign the factors of a reported bug, it is still a time-consuming manual process. Therefore, it is necessary to develop an approach to predict the bug severity and priority. Essentially, both the priority and severity predictions belong to the classification problem, which means that a bug report should be arranged to a category labeled by severity level or priority level. However, we need to address the challenge on how to utilize the textual contents and features of a new bug report to classify it into a correct category (i.e., priority level or severity level). Thus, the selection of features and classifiers become the key to address the problem. Various approaches were proposed to predict the priority [18–22] and the severity [23–30] of reported bugs. Since all approaches adopted machine-learning algorithms to conduct the prediction tasks, we consider the different implement means to categorize them into two classes, including coarse-grained prediction and find-grained prediction.

Coarse-grained prediction For coarse-grained prediction, the approaches do not predict each priority and severity levels. In other words, these previous studies only predicted a rough classification (e.g., 'non-severe' and 'severe') for each new bug report.

Alenezi and Banitaan in [21] adopted Naive Bayes, Decision Trees [86] and Random Forest [87] to execute the priority prediction. They used *tf* to get the weight of words in each bug report as the first feature set, and introduced component, operating system and severity as the second feature set to perform three machine-learning algorithms. The results of the evaluation experiments showed that the algorithms using the second feature set performed better than using the first feature set, and also demonstrated that Random Forests and Decision Trees outperformed Naive Bayes.

In [23], Lamkanfi *et al.* proposed a severity prediction approach by analyzing the textual description of a given bug report. In this study, they divided the severity levels *trivial* and *minor* into 'severe', while grouped *major*, *critical* and *blocker* into 'non-severe'. Then they used Naive Bayes classifier which is based on the probabilistic occurrence of terms to categorize the given bug report into 'severe' or 'non-severe'. The evaluation results indicated that this Naive Bayes-based classification method can reach a reasonable accuracy for Mozilla, Eclipse and GNOME. As a follow-up work, Lamkanfi and colleagues [24] tried to compare four text mining algorithms namely Naive Bayes [88], Naive Bayes Multinomial [89], KNN [90] and SVM [91] for predicting the severity of a reported bug. They found that Naive Bayes Multinomial reached the best performance than other algorithms.

The studies [26, 27, 29] adopted the different ways to improve the performance of coarse-grained severity prediction. In [26], Yang *et al.* utilized feature selection schemes such as Information Gain, Chi-Square and Correlation Coefficient to improve the performance of severity prediction using Multinomial Naive Bayes classification approach. The experimental results showed that these feature selection schemes can effectively extract potential severe and non-severe indicators and thus improve the prediction performance in over half the cases. In [27], Bhattacharya *et al.* proposed a graph-based characterization of a software system to facilitate the task of severity prediction. In detail, they used *NodeRank* to verify critical functions and modules which are likely to indicate high-severity bugs. The major function of *NodeRank* is to measure the relative importance of a node (i.e., function or module) in the module collaboration graphs. Thus they can identify how critical a function or module is. In [29], Xuan *et al.* addressed the problem of developer prioritization to rank the contributions of developers based on social network, then they used the results of the prioritization to improve the performance of three software maintenance tasks that include severity prediction. By combining the features adopted by [23], they used Naive Bayes to execute the prediction task. The results showed that the developer prioritization can improve the performance by $\sim$1% for precision, recall and F-measure.

Fine-grained prediction Different from coarse-grained prediction, fine-grained prediction approaches can predict the priority and severity levels.

Yu *et al.* [18] proposed a new approach to predict the priority-level based on Neural Network [92]. In detail, they first manually classified the bugs, followed by automated learning by artificial neural network (ANN), then predicted the priority levels with the learned ANN. Moreover, they extracted nine attributes, including milestone, category, module, severity, main workflow, function, integration, frequency and tester, from bug reports as the inputs of ANN model. The result of evaluation showed that this prediction approached based on ANN performed better than Naive Bayes. For example, for the priority-level $P1$, the F-measure of ANN achieves 88%, which is much better than 34.7% produced by using Naive Bayes.

Kanwal and Maqbool [19] used Naive Bayes and SVM to present a comparison with evaluate which classifier performs better in terms of accuracy. The evaluation results on all priority levels showed that the performance of SVM is better than Naive Bayes when adopting text features (i.e., summaries and long descriptions of bug reports), while for categorical features such as component, severity and platform Naive Bayes performed better than SVM. The highest accuracy is achieved with SVM when combining the text and categorical features. For the category including the combining features, the precision of SVM reached up to 54%, which is better than Naive Bayes (40%).

Sharma *et al.* [20] applied SVM, Naive Bayes, KNN and Neural Network to predict the priority of the newly arrived bug reports. They calculated the term weight in each bug report using $tf * idf$, then utilized these machine-learning algorithms to classify the given bug reports into different categories (i.e., priority levels from P1 to P5). They performed cross-validation in the evaluation experiment and demonstrated that the accuracy of different machine-learning algorithms reached $<$70% except for the Naive Bayes.

By analyzing the multiple factors-temporal, textual, author (e.g., reporter), related report, severity and product that potentially affect the priority of bug reports, Tian and her colleagues [22] extracted them as features to train a discriminative model that can verify which priority level a new bug report belongs to. In detail, the proposed approach introduced linear regression to capture the relationship between the features and the priority levels, then they employed a thresholding approach to adjust the different thresholds to decide the priority levels. The experimental results showed that the proposed approach outperformed the method presented in [25], which was used to predict the priority of bug reports.

Menzies and Marcus [25] proposed a fine-grained bug severity prediction approach. They utilized InfoGain [93] to rank all terms appearing in the bug reports based on the term weight calculated by $tf * idf$, then a data miner was adopted to learn rules that predict the severity levels using the top-k terms. The case study results showed that the developed prediction tool can effectively predict the bug severity levels.

Tian, Lo and Sun's study [28] also focuses on fine-grained severity prediction. To realize this goal, an IR-based nearest neighbor solution was proposed. They first calculated the similarity of different bug reports and based on this similarity the historical bug reports that are most similar to a new bug report are identified. These reports are marked as duplicates which can help to identify the different severity level of the given bug. In this work, authors adopted $BM25F_{ext}$ [94] which is an extension version of $BM25F$ to compute the similarity between the new bug report and the historical reports. In detail, $BM25F$ is developed to compute the textual similarity of a short document (which is marked as query) and a longer document. However, bug reports are longer documents. In order to calculate the similarity of two bug reports, Tian *et al.* adopted $BM25F_{ext}$ and then they found the duplicate bug reports as the nearest neighbors to predict the severity level of the new bug report. Compared with the severity prediction algorithm proposed by Menzies and Marcus [25], the proposed approach exhibits a significant improvement.

For the same reason, Yang *et al.* [30] developed a new algorithm to predict the severity of a new reported bug from another angle. They utilized topic model and multiple factors such as priority, component and product to find the historical bug reports which have the strong relevance with the given bug. Then KNN was adopted to predict the severity levels. The evaluation results showed that this algorithm can improve the performance of other studies, namely single KNN and Naive Bayes.

Comparison of priority prediction approaches: A detailed comparison of priority prediction approaches [18–22] is provided in Table 4. Note that the studies [18–20, 22] executed the fine-grained prediction for priority levels of bug reports while the coarse-grained prediction that only considered two category 'high priority' and 'low priority' implemented in [21]. The fine-grained prediction is actually a multi-class classification task and the coarse-grained prediction is a two-class

TABLE 4. Comparison of priority prediction approaches.

Works	Mechanism	Method summary	Advantages	Disadvantages
[21]	Coarse-grained prediction	Adopted Naive Bayes, Decision Trees and Random Forest to execute priority prediction.	Demonstrated that the second feature set (i.e., factors of bug reports) outperformed the first feature set (i.e., term weight) by 17.8% average F-measure for high-class priority of given bugs in Eclipse and Firefox projects.	Only two projects (i.e., Eclipse and Firefox) are utilized.
[18]	Fine-grained prediction	Used Neural Network to predict the priority levels.	Reached up to 81.4% average F-measure, which is better than Naive Bayes (47.9%) for bug reports at Hospital Information System, Software Development Tools, ERP system and Word Processing Software from an international medical device maker.	Needs more features.
[19]	Fine-grained prediction	Adopted Naive Bayes and SVM to present a comparison to evaluate which classifier performs better in terms of accuracy.	SVM with the combining features achieved 54% precision, which is better than Naive Bayes (40%) for the Eclipse Platform project.	Needs more features.
[20]	Fine-grained prediction	Utilized SVM, Naive Bayes, KNN and Neural Network to predict the priority of the newly arrived bug reports.	Demonstrated that the accuracy of different machine-learning techniques (except Naive Bayes) can successfully predict the priority of $<$70% bugs from Eclipse and OpenOffice projects.	The prediction accuracy depends on the quality of bug report summaries.
[22]	Fine-grained prediction	Extracted multi-factors (e.g., severity, product, etc.) as features to implement linear regression and thresholding algorithm for predicting the priority levels of a new bug report.	Outperformed SVM-MultiClass by a relative improvement of 58.61% for Eclipse project.	Needs more features.

classification problem. Both of them adopted machine-learning algorithms such as SVM, Naive Bayes to predict the priority levels. We found that the feature selection has the significant effect to the performance of prediction. For example, in [21], the adopted algorithms utilizing the second feature set that includes component, operating system and severity performed better than them used the first feature set that contained the weighed words extracted from bug reports; in [19], SVM with the combing features including text features and categorical features performed better than Naive Bayse, however, when only using the categorical features, SVM performed worse than Naive Bayes.

Comparison of severity prediction approaches: A detailed comparison of severity prediction approaches [23–30] is provided in Table 5. Note that [23, 24, 26, 27, 29] focused on coarse-grained prediction, while [25, 28, 30] developed fine-grained prediction algorithms. For the coarse-grained prediction approaches, the studies [26, 27, 29] adopted the different mechanisms such as feature selection, graph-based analysis

TABLE 5. Comparison of severity prediction approaches.

Works	Mechanism	Method summary	Advantages	Disadvantages
[23]	Coarse-grained prediction	Utilized Naive Bayes classifier to categorize bug reports into 'non-severe' and 'severe', respectively.	Predicted the severity with a reasonable accuracy (both precision and recall ranged between 65–75% with Mozilla and Eclipse, and 70–85% for GNOME)	Relies on the presence of a causal relationship between the contents of the fields in the bug report and the severity of the bugs.
[24]	Coarse-grained prediction	Utilized Naive Bayes, Naive Bayes Multinomial, KNN and SVM to predict the bug severity and compared their performance.	Demonstrated that the average accuracy of Naive Bayes Multinomial reached up to 73% and 85.7% for all products of Eclipse and GNOME, respectively, which performed better than Naive Bayes, SVM and KNN.	Relies on the presence of a causal relationship between the contents of the fields in the bug report and the severity of the bugs.
[26]	Coarse-grained prediction	Utilized feature selection schemes to improve the performance of severity prediction using Multinomial Naive Bayes.	Improved the prediction performance in over half the cases at Mozilla and Eclipse.	The feature selection schemes do not consider the semantic relations.
[27]	Coarse-grained prediction	Used graph-based analysis to facilitate severity prediction.	Worked at both function and module level and predicted bug severity before a bug report is filed.	Additional cost for building the graphs.
[29]	Coarse-grained prediction	Utilized developer prioritization to enhance the performance of severity prediction.	Improved the accuracy of severity prediction only using the machine-learning algorithm by 1% at Eclipse and Mozilla.	Additional cost for building the social network to prioritize the developers.
[25]	Fine-grained prediction	Used InfoGain to rank all the terms in the data set based on the term weight $tf * idf$ and then adopted a data miner to learn rules that predict the severity levels using the top-k terms.	Achieved an average precision, recall and F-measure of up to 50.5, 80 and 50.5%, respectively, for all severity levels of bugs in NASA by using little domain knowledge.	Relatively lower accuracy for Severity 4 (8% precision) and used only one data set (i.e., NASA).
[28]	Fine-grained prediction	Adopted $BM25F_{ext}$ to calculate the similarity between bug reports for finding the most similar reports with a new bug report and used them as the nearest neighbors so that the severity level of the given bug can be predicted.	Achieved a precision, recall and F-measure of up to 72, 76 and 74% for predicting a particular class of severity labels, which performed better than [25] at Eclipse, OpenOffice and Mozilla.	Needs to adjust the value of parameter k (the number of nearest neighbors) while using different data set.
[30]	Fine-grained prediction	Employed topic model and multiple factors (e.g., component) to predict the severity levels.	Produced higher accuracy (<70%) than traditional classifiers such as KNN (10–14% improvement) and Naive Bayes (3–8% improvement) for Eclipse and Mozilla.	Needs to adjust the value of parameter k (the number of topics) while using different data set.

and developer prioritization to improve the prediction performance by only using the traditional two-class machine-learning algorithms. For the fine-grained prediction algorithms, [28] reached a better performance than [25] which introduced only one data set for evaluation while [30] performed better than KNN and Naive Bayes. The authors [28, 30] evaluated their methods in three different open source projects, respectively, however, these algorithms also need to adjust the corresponding parameters for adapting to the different data sets.

2.3.2. Reopened/blocking prediction approaches

In some cases, the bugs marked as 'closed' can be reopened due to many reasons. For example, if a bug was not incorrectly fixed and reappeared, in this situation, the status of this bug can be regarded as 'reopened'. Blocking bugs are software defects that prevent other defects from being fixed. Under the environment, the developers cannot fix the bugs because the components that they are fixing depend on other components that have unresolved bugs. Obviously, the reported bugs marked as 'reopened' or 'blocking' increase the bug-fixing cost and fixers' workload, therefore some scholars devoted to study the reasons why the bugs can be reopened or marked as 'blocking', and predict whether a new given bug is reopened or blocking. For example, Zimmermann *et al.* [95] characterized the overall process of reopen process by conducting a survey to a large population of experienced developers on the fundamental reasons for bug reports, and built a statistical descriptive model to identify statistically the most important factors affecting reopened bugs. But their study did not predict whether an individual bug is reopened or not. Garcia and Shihub [34] found that blocking bugs take approximately two to three times longer to be fixed compared with non-blocking bugs, and built the prediction models for identifying the blocking bugs. In our reviewed papers, we found that the studies [29, 31–33] proposed a prediction model for reopened prediction while the works [34, 35] devoted to predict the blocking bugs. We do not classify these studies like other tasks, because all studies utilized machine-learning algorithms with the features of bug reports to predict the reopened or blocking bugs. They have the same implementation way (i.e., two-class classification) and the similar algorithms (i.e., machine-learning algorithms), thus there is no need for categorizing them.

We introduce the above-mentioned studies and show the comparison among them as following subsections.

The prediction approaches for reopened bugs In order to predict whether a new bug will be reopened or not, the machine learning algorithms such as SVM, decision tree are utilized to build a two-class discriminative model for classifying the given bugs. Moreover, the features (e.g., component and product) of bug reports are adopted to enhance the performance of prediction.

Shihab *et al.* [31] first extracted the features of bug reports to structure four dimensions, including the work habits dimension (e.g., the initial closed time), the bug report dimension (e.g., component), the bug fix dimension (e.g., fixing time) and the people (team) dimension (e.g., the fixers' experience), then they created decision trees based on these dimensions to predict whether a bug will be reopened. The predictions model can reach up to 62.9% precision and 84.5% recall.

Xuan *et al.* [29] added two factors, i.e., the reporter priority score and the fixer priority score, into the people dimension presented in [31]. Then they implemented the same approach proposed by Shihab *et al.*, the results showed that these two factors brought the slight improvement for the performance of the method claimed in [31]. Xuan and his colleagues explained that the small size of training set may limits the predictive ability of two factors so that the performance improvement is not significant.

Xia *et al.* [32] evaluated the effectiveness of various supervised learning algorithms (e.g., KNN, SVM, Decision Table, Bagging and Random Forest) to predict whether a bug report will be reopened. The experimental results showed that Bagging and Decision Table achieved the best performance. They reached up to accuracy score of 92.9 and 92.8%, respectively. These results improved F-measure of the approach proposed by Shihab *et al.* [31] by up to 23.5%.

In the following work, Xia and his group proposed *ReopenPredictor* [33], which combined the multiple type of features (i.e., description features, comment features and meta features) and adopted the corresponding classifiers, including a description classifier, a comment classifier and a meta classifier to achieve a higher performance than previous approaches such as [31]. The experimental results showed that *ReopenPredictor* achieved an improvement in the F-measure of the prediction approach proposed by Shihab *et al.* by 33.33, 12.57 and 3.12% for Eclipse, Apache HTTP and OpenOffice, respectively.

A detailed comparison of the prediction approaches [29, 31–33] for reopened bugs is provided in Table 6. Note that all of approaches adopted machine-learning algorithms to predict the reopened bugs, thus the selection of machine-learning algorithms and features of bug reports can necessarily affect the prediction performance. For example, Xia *et al.* [33] proposed *ReopenPredictor* combined the multiple features and the corresponding classifiers to predict whether a given bug will be reopened or not so that it performed better than the approach proposed by Shihab *et al.* [31].

The prediction approaches for blocking bugs: Similar to the prediction for reopened bugs, predicting blocking bugs also adopted machine-learning algorithms and features in the previous studies [34, 35] because this prediction task is also a two-class classification problem.

In [34], Garcia and Shihab extracted 14 features such as the textual description of bug reports and the bug location from the bug tracking systems to build decision trees for each project to predict whether a bug will be a blocking bug or not. Moreover, they analyzed these features to find which features have the best influence to indicate the blocking bugs. As the results, the proposed method achieved F-measure of 15–42%, and the features, including comment text, comment size, the number of developers and the reporter's experience, were found as the most important indicators for identifying the blocking bugs.

Xia *et al.* [35] proposed *ELBloker* to identify blocking bugs. This method first randomly divided the training data into multiple disjoint sets, and for each disjoint set, a classifier is built

TABLE 6. Comparison of prediction approaches for reopened bugs.

Works	Mechanism	Method summary	Advantages	Disadvantages
[31]	Machine-learning-based prediction with multi-features	Utilized decision trees based on four dimensions including 22 features to predict reopened bugs.	Implemented effective prediction for reopened bugs by 62.9% precision and 84.5% recall at the Eclipse Platform project.	Needs more features.
[29]	Machine-learning-based prediction with multi-features	Utilized developer prioritization to improve the prediction for reopend bugs.	Improved the prediction accuracy of the approach proposed by Shihab *et al.* [31] by up to 0.7% at Eclipse.	Needs to extract the features by prioritizing developers and slight improvement for prediction performance.
[32]	Machine-learning-based prediction with multi-features	Evaluated the effectiveness of various supervised learning algorithms to predict reopened bugs.	Improved F-measure of the approach proposed by Shihab *et al.* [31] by up to 23.5% at the Eclipse Platform project.	Additional evaluation cost.
[33]	Machine-learning-based prediction with multi-features	Utilized multiple features and corresponding classifiers to predict the reopened bugs.	Achieved an improvement in the F-measure of the prediction approach proposed by Shihab *et al.* by 33.33, 12.57 and 3.12% for Eclipse, Apache HTTP and OpenOffice, respectively.	Needs more features.

based on random forest algorithm. Then it combined these multiple classifiers to determine an appropriate imbalance decision boundary to distinguish blocking bugs from non-blocking bugs. *ELBloker* improved the F-measure over the approach proposed by Garcia and Shihab [34] by 14.69%.

A detailed comparison of the prediction approaches [34, 35] for blocking bugs is provided in Table 7. Note that combining more features and classifiers can improve the prediction performance. For example, *ELBloker* [35] combined the multiple classifiers and utilized more features to enhance the prediction model so that it can produce the better performance than the prior method presented in [34] by up to 14.69% F-measure. However, the better approach may increase the running cost due to more extracted features and utilized classifiers.

3. BUG TRIAGE

Triagers are responsible to assign the new bug reports to the right assignees for fixing the given bugs. Automatic bug triage can reduce the probability of re-assignment and reduce triagers' time by recommending the most appropriate assignees. However, it throws out a challenge on how to verify the candidate assignees and how to rank them. Existing approaches utilize a series of approaches (e.g., machine-learning algorithms, social network metrics) to quantify developers' experience so that they can rank the assignees and find the most appropriate one. In this section, we survey the related works and show the advantages and disadvantages.

By employing multiple techniques such as machine-learning algorithms and social network analysis, the previous studies [29, 30, 36–53] can achieve the purpose of automatic assignee recommendation. According to the different techniques, we classify these studies into five category such as machine learning-based recommender, expertise model-based recommender, tossing graph-based recommender, social network-based recommender and topic model-based recommender, which are detailed in the following subsections, respectively.

3.1. Machine-learning-based recommender

Since machine-learning algorithms can learn from the data, previous studies utilized machine-learning algorithms, such as Naive Bayes and SVM, to decide the most appropriate developer for fixing the given bug.

As a pioneering study, Čubranić and Murphy [37] proposed an automatic bug triage approach to recommend the best developers for fixing the given bugs. In this work, the problem of assigning developers to fix bugs was treated as an instance of text classification. For this classification problem, each assignee was considered to be a single class and each bug report was assigned to only one class. Therefore, authors used a traditional

TABLE 7. Comparison of prediction approaches for blocking bugs.

Works	Mechanism	Method summary	Advantages	Disadvantages
[34]	Machine-learning-based prediction with multi-features	Utilized decision trees based on 14 features to predict blocking bugs.	The prediction models achieved F-measures of 15–42%, which is a two- to four-fold improvement over the baseline random predictors for the bug reports extracted from Chromium, Eclipse, FreeDesktop, Mozilla, NetBeans and OpenOffice.	Lower percentage of blocking bugs causes the classifier not to learn to predict the blocking bugs very well.
[35]	Machine-learning-based prediction with multi-features	Combined the multiple classifiers to determine an appropriate imbalance decision boundary to identify blocking bugs.	Improved the prediction performance of the approach proposed by Garcia and Shihab [34] by up to 14.69% F-measure at Chromium, Eclipse, FreeDesktop, Mozilla, NetBeans and OpenOffice.	Needs to extract more features and build the multiple classifiers for all disjoint sets.

machine-learning algorithm-Naive Bayes to determine which category a new bug report belongs to and then recommend the best assignee. However, the accuracy is not very high (only 30% bug assignees were predicted correctly).

In order to improve the performance of automatic bug triage, Anvik and colleagues in their article [38] and the following extended work [44] employed several different machine-learning algorithms (e.g., Naive Bayes, SVM, C4.5) to recommend a list of appropriate developers for fixing a new bug. They demonstrated that SVM performed better than others on their data sets from open source projects such as Eclipse, Mozilla Firefox. In detail, in [38], they have reached precision levels of 57 and 64% on the Eclipse and Firefox projects, respectively. In [44], they improved the precision by up to 97% for Eclipse and 70% for Firefox by utilizing component-based developer recommender.

Lin *et al.* [40] proposed two automatic bug triage approaches, namely text information-based developer recommender and non-text information (e.g., bug type, bug class and priority)-based recommender. Note that this is a first work to conduct developer recommendation using Chinese text. In detail, they adopted SVM to implement Chinese text-based automatic bug triage, and utilized C4.5 decision tree to perform non-text information-based automatic bug triage. The results showed that the accuracy of the text-based approach achieved 63%, which is close to 67% accuracy of manual approach for bug assignment. In addition, the accuracy of the non-text approach reached up to 77.64%, which outperforms the text-based approach and the manual approach.

Ahsan *et al.* [41] reduced the dimensionality of the obtained term-to-document matrix by using feature selection and Latent Semantic Indexing (LSI) [96]. They utilized several machine-learning algorithms to recommending the bug fixers, and the results showed that the best obtained bug triage system is based on LSI and SVM. The average precision and recall values reached up to 30 and 28%, respectively.

Xuan *et al.* [42] proposed a semi-supervised text classification approach for bug triage. In order to avoid the deficiency of labeled bug reports in existing supervised approaches, this method enhanced Naive Bayes classifier by utilizing expectation–maximization based on the combination of labeled and unlabeled bug reports to recommend the appropriate bug fixers. The experimental results showed that the accuracy can be improved by up to 6% by comparing with the recommender using original Naive Bayes classifier.

Zou *et al.* [47] utilized the feature selection algorithm and the instance algorithm to reduce the size of training set by removing the noisy data, respectively. Then they adopted Naive Bayes to execute the task of automatic bug triage. The experimental results showed that the feature selection can improve the performance using original Naive Bayes by up to 5%, but the instance algorithm lowered the performance even though it can reduce the size of training set.

Xia *et al.* [52] proposed an accurate approach called DevRec for recommending the bug fixers. DevRec combined two kinds of analysis, including bug reports-based analysis (BR-based analysis) and developer-based analysis (D-based analysis). Based on Multi-Label KNN, BR-based analysis can find the k-nearest bug reports to a new given bug by using the features (i.e., terms, product, component and topics) of bug reports. For D-based analysis, Xia and his colleagues measured distances between a developer and a term, a product, a component and a topic. Then they built a composite model to combine the results of BR-based analysis and D-based analysis for executing automatic bug triage. The experimental results showed that DevRec improved the recall values of Bugzie [45] and DREX [46] by 39.39 and 89.36%, respectively when recommending top-10 developers.

3.2. Expertise model-based recommender

Building the expertise model can capture the developers' expertise on historical fixing activities. Matter *et al.* [39] modeled developers' expertise using the vocabularies found in their source code files and compare them to the terms appearing in corresponding bug reports. In detail, they adopted cosine measure to compute the similarity between two term vectors. The evaluation results showed that this method can achieve 33.6% precision when recommending top-1 assignee and 71% recall while recommending top-10 assignees.

Tamrawi *et al.* [45] proposed Bugzie, an automatic bug triaging tool based on fuzzy set and cache-based modeling of the bug-fixing expertise of developers. Bugzie extracted the multiple technical aspects of the software systems, each of which is associated with technical terms. For each technical term, it used a fuzzy set to represent the developers who have able to fix the bugs related to the corresponding aspect. In this work, a candidate developer's membership score was used to reflect the fixing correlation of the developer. For a new bug report, Bugzie combined the fuzzy sets corresponding to the terms and ranked the developers based on their membership scores. According to the reported evaluation results, Bugzie performed better than other approaches [37–39, 43]. For example, for Eclipse, in term of top-5 recommended developers, Bugzie spent only 22 minutes and achieved 72% accuracy, which is 49 times faster and relatively 19% more accurate than SVM [38], which is second best model.

Servant *et al.* [49] developed a developer recommender to fix the given bugs. This tool consists three components, including bug localization, history mining to find which developers changed each line of source code and expertise assignment to map locations to developers. In detail, the expertise assignment algorithm combined the history information of code changes and the diagnosis information about the location of bugs to provide a ranked list of developers in terms of expertise in these locations. As a result, the proposed developer recommender can achieve 81.44% accuracy when recommending top-3 developers.

3.3. Tossing graph-based recommender

Jeong *et al.* [36] proposed a tossing graph model to capture bug tossing history. By revealing developer networks which can be used to discover team structure, it can find suitable assignees for a bug-fixing task. In this study, a tossing process starts with the first assignee and moves from one assignee to another until it reaches the final assignee. Each move is called a tossing step and a set of tossing steps of a bug report is called a tossing path. Suppose an original tossing graph $A \rightarrow B \rightarrow C \rightarrow D$, authors' goal is predicting a path to D with fewer steps. They tried to reduce the tossing graph to avoid unnecessary tossing for timely bug fixing. The experiments on Eclipse and Mozilla data sets demonstrated the tossing graph model can improve the accuracy of automatic bug triage using machine-learning algorithms such as Naive Bayes and Bayesian Network [97] only.

In [43], Bhattacharya and Neamtiu improved the accuracy of bug triage and reduced tossing path lengths by utilizing several techniques such as refined classification using additional attributes and intra-fold updates during training, a precise ranking function for recommending potential tosses in tossing graphs, and multi-feature tossing graph. The experimental results on Mozilla and Eclipse showed that the recommendation accuracy achieved 84 and 82.59%, respectively. Moreover, the proposed method can reduce the length of tossing paths by up to 86% for correct recommendations.

In the following work [48], Bhattacharya *et al.* utilized Naive Bayes coupled with product-component features, tossing graphs and incremental learning to enhance the developer recommender. As the results, the accuracy achieved up to 85% for Mozilla and 86% for Eclipse, which is higher than their early study [43]. In addition, this claimed method reduced the length of the tossing paths by up to 86%, which kept the same value with [43].

3.4. Social network-based recommender

In recent years, social network technique [98] has been introduced to find the potential experienced assignees for fixing each new reported bug. It is a better way to analyze the relationship between developers in the bug-fixing process. Generally, the commenting activities in this process become the object in the social network. Figure 4 shows an example of social network revealing the commenting activities between developers (i.e., assignees and commenters). Note that there are four nodes which represent four developers A, B, C and D. The links represent the commenting activities between developers. In this example, developers discuss how to resolve the given bugs by posting the comments. The number on each link captures the number of comments posted by the commenter towards the assignee who was assigned to fix the reported bugs. For instance, the number '3' on the link from developer A to B means that B posted the comments three times on the bug reports assigned to A. In some special cases, developers may post the comments to the bug reports assigned to themselves. We called these cases as 'self-links' (e.g., 1 time for developer C). When analyzing the social network, the number of 'self-links' are usually not considered. Depending on the analysis of the commenting activities among developers, we can understand the developers' experience on the fixing tasks for the specific bugs so that the potential assignees for a new

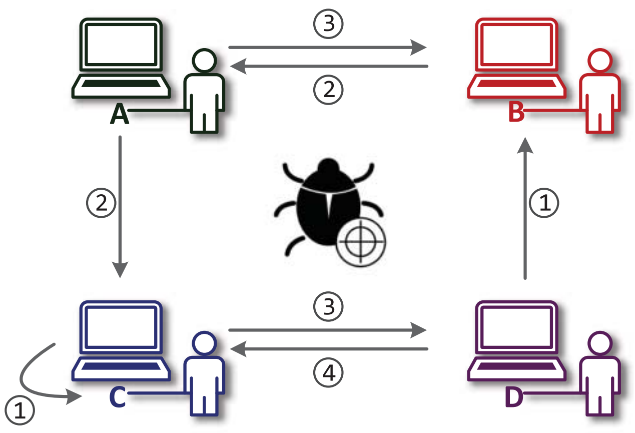


FIGURE 4. An example of social network between developers.

reported bug can be found. Both [29, 46] are representative studies which employed the social network technique.

In [46], Wu and colleagues adopted KNN to search the historical bug reports that are similar to a new bug report, and then extracted the developers who posted the comments to these bug reports as the candidates. As a final step, they used simple-frequency and six social network metrics, including in-degree centrality, out-degree centrality, PageRank, betweenness centrality and closeness centrality. Thereinto, simple-frequency was used to compute the candidates' participation records in the comments; in-degree centrality was defined as the number of links (commenting activities) directed to the node (developer); out-degree centrality was defined as the number of links that the node directs to other nodes; degree centrality was defined as the number of links that the node has; PageRank was applied to measure the score for each node; betweenness centrality was introduced to compute the number of shortest paths from all vertices to all others that pass through that node and closeness centrality was introduced to compute the number of shortest paths between a vertex and all other vertices reachable from it. By evaluating simple-frequency and all social network metrics, authors found that simple-frequency and out-degree showed the best performance.

Xuan *et al.* [29] achieved developer prioritization via social network analysis to improve the performance of automatic bug triage using SVM or Naive Bayes only. In the prioritization process, they gave a score s_i for each developer d_i and ranked all the developers based on these scores. By analyzing the social network between developers in the commenting activities, they calculated a weight ω_{ji} which denoted the number of all the comments in a link from developer d_j to d_i and the out-degree o_j of developer d_j. By ranking the developers who participated the commenting activities in the past bug-fixing tasks, Xuan and his colleagues utilized the prioritization results to redo the developer recommendation using Naive Bayes and SVM, respectively. The results showed that developer prioritization via social network analysis can further improve the accuracy of automatic bug triage using machine-learning algorithms only.

3.5. Topic model-based recommender

Topic model can help to find historical bug reports similar to a new bug report, which share the same topic(s). The scholars expect to introduce topic model for improving the accuracy of automatic bug triage. Studies [30, 50, 51, 53] adopted topic model to recommend the best developers for fixing the given bugs.

In [50], Xie *et al.* proposed a fixer recommender called DRETOM, which used Stanford Topic Modeling Toolbox (TMT)[3] to build topic model for grouping the bug reports which share the same topic(s). Given a new bug report, it is easy to know which topic(s) this bug belongs to. Then they analyzed the developers' interests and experiences on the bug reports belonging to the corresponding topic(s) in the past fixing records so that the proposed algorithm can work well for recommending the appropriate fixers. The experimental results showed that the proposed method performed better than machine-learning algorithms (i.e., SVM and KNN) and social network analysis-based recommender.

LDA is a generative probabilistic model for arranging the discrete data into different topics [82]. Naguib *et al.* [51] adopted LDA to cluster the bug reports into topics. Then they created the activity profile for each developer of the bug tracking repository by mining history logs and bug report topic models. An activity profile consists of two parts, including developer's role and developer's topic associations. By utilizing activity profile and a new bug's topic model, they proposed a ranking algorithm to find the most appropriate developer to fix the given bug. The result showed that the claimed approach can achieve an average hit ration of 88%.

Yang *et al.* [30] also used TMT to generate the topic model for verifying the topic(s) that a new bug report belongs to. Then, they extracted candidate developers from historical bug reports on the same topic(s) and same multi-features such as product, component, priority and severity with the given bug report. Next, based on a social network analysis between these candidates, Yang and his colleagues captured their commenting activities and committing activities which reflected the changes in the source code files related to the given bug so that the developed recommender can effectively execute automatic bug triage. The evaluation results showed that it outperformed the recommender proposed in [50] and the social network metric out-degree [46].

Zhang *et al.* [53] utilized LDA to extract the topics from historical bug reports. By capturing the developers' behavior on historical bug reports belonging to a same topic with the given bug report, they can verify whether the developer has

[3] http://nlp.stanford.edu/software/tmt/tmt-0.4/

interest in this topic. In addition, they analyzed the relations between the developers (i.e., bug fixers and commenters) and the bug reporter whose bug reports attracted the most number of comments. By combining the correlation score and the relation score, the proposed approach can improved F-measure of DRETOM [50] and Activity Profile [51] for Eclipse project by up to 3.3 and 16.5%, respectively.

3.6. Comparison of automatic bug triage approaches

The comparison of different approaches [29, 30, 36–53] proposed for automatic bug triage is summarized in Table 8. Note that using feature selection technique or composite model can enhance the machine- learning algorithms and consequently improve the accuracy of automatic bug triage. For example, Zou *et al.* [47] utilized feature selection mechanism to enhance the recommender using Naive Bayes by up to 5% accuracy, and DevRec [52] proposed a composite model combining BR-based analysis and D-based analysis for achieving higher performance than DREX [46] and Bugzie [45]. Moreover, introducing the new popular techniques such as social network techniques and topic model can improve the accuracy of developer recommendation using machine-learning algorithms only. For instance, social network analysis-based recommender [29] performed better than SVM and Naive Bayes, and DRETOM [50] showed a better performance than SVM, KNN, out-degree and degree centrality.

4. BUG FIXING

Bug fixing is the ultimate purpose of bug resolution. Assignees should find the location of the given bugs, and then produce the patches in the fixing process. Bug localization and automatic patch generation can reduce the assignees' workload and the fixing time. In this section, we summarize the previous studies in these areas.

4.1. Survey on bug localization approaches

When a new bug report is assigned to an assignee for fixing, the assignee needs to find where the bug is. Obviously manual bug localization increases the assignees' workload. Therefore, the automatic bug localization method needs to be developed for avoiding the manual process. The major challenge is how to verify the relationship between the source code files and a new reported bug. IR-based models can be adopted to search the relevant source files and rank them to find the correct location file. In these IR-based approaches, each bug report was treated as a query, and the source code files to be searched comprise the document corpus. A ranked list of candidate source code files where the bug may appear can be returned. Besides IR-based models, some studies [61–66] also adopted other information or model to improve the performance of bug localization. We introduce these different studies as following subsections.

4.1.1. IR-based bug localization

By utilizing IR-based techniques, the existing studies can find the candidate source code files that are related to a new bug, and then rank them to verify which file(s) the given bug appears. Lukins *et al.* [54, 55] utilized LDA to execute the source code retrieval for bug localization. In this study, they adopted GibbsLDA++ to find the topics from the source code files as well as get the term-topic and topic-document (i.e., bug report) probability distributions. When a new bug report arrives, its title and description are extracted as a query to verify which topic the bug report belongs to. Thus, a method (or a class) in the source code file is relevant to the query if it has the high probability of generating the terms in the query. By calculating the similarities between the query and each method (or class) sharing the same topic, it is easy to know which topic the query belongs to so that authors can provide a ranked list to recommend corresponding methods (or classes) for locating the given bug. In [54], the method returned a relevant method in the top-10 for 77% of 35 bugs in the open source project-Rhino. In the extended version [55], Lukins *et al.* analyzed 322 bugs across 25 versions of two software systems that include Eclipse and Rhino. The evaluation results showed that over one-half of the bugs were located successfully. Moreover, the search space (i.e., the number of methods that a debugger must check) in Eclipse was reduced to <0.05% of the total methods and in Rhino was reduced to <0.5%.

Rao and Kak [56] evaluated different IR methods' performance of bug localization, including smoothed Unigram Model [99], VSM, Latent Semantic Analysis Model (LSA) [100], LDA and Cluster Based Document Model (CBDM) [101]. The experimental results showed that smooth UM and VSM are more effective at retrieving the buggy files of the given bugs.

Different configurations of IR models (e.g., LDA, VSM, and LSI) can affect the performance of bug localization. For example, different number of topics in LDA can affect the results of IR [30, 53, 54]. Thomas *et al.* [57] evaluated the influence on the performance of bug localization by adopting different configurations of three IR-based classifiers (i.e., LDA, VSM and LSI) and one Entity Metrics-based classifier [102] so that they can find the best one for each classifier. Then they combined the best-performing classifiers based on the logic that classifiers using different data sources as input. The experimental results showed that the combination can achieve the improvements ranged from −2 to 12%, depending on the different project and combination method used.

Davies *et al.* [58] measured the similarity between the text used in a new bug report and the text of existing fixed bug reports to enhance the previous localization approach by only calculating the similarity between the given bug report and the source code. The experimental results showed that using the measurement results between bug reports (i.e., *BUG*) can improve the performance by only measuring the similarity between the given bug report and the source code (i.e., *SOURCE*). Specifically, the

TABLE 8. Comparison of automatic bug triage approaches.

Works	Mechanism	Method summary	Advantages	Disadvantages
[37]	Machine-learning-based recommender	Utilized Naive Bayes to verify which developer is an appropriate assignee.	Adopted fewer features (e.g., summary and description) of bug report.	Relatively lower accuracy (30%) for Eclipse bug reports.
[38]	Machine-learning-based recommender	Used Naive Bayes, SVM, and C4.5 to predict the potential assignee.	Demonstrated that SVM-based recommender achieved precision levels of 57 and 64% on the Eclipse and Firefox projects, respectively, which performed better than Naive Bayes and C4.5.	Achieved only 8% precision for gcc project when recommending one developer.
[44]	Machine-learning-based recommender	Used component-based technique to enhance the developer recommender presented in [38].	Improved the precision value of [38] by up to 97% for Eclipse and 70% for Firefox.	Needs to extract the feature ‘component’.
[40]	Machine-learning-based recommender	Proposed Chinese text information-based developer recommender and non-text information-based developer recommender.	Demonstrated that the non-text information recommender (77.65% accuracy) outperformed the text-based approach and the manual approach for the bug reports from the SoftPM project.	Needs more features.
[41]	Machine-learning-based recommender	Used feature selection and LSI to enhance the machine learning algorithms such as SVM to execute the developer recommendation.	Demonstrated LSI and SVM performed best among all machine-learning algorithms.	Relatively lower precision and recall values by up to 30 and 28%, respectively, for the Mozilla project.
[42]	Machine-learning-based recommender	Utilized expectation-maximization based on the combination of labeled and unlabeled bug reports to enhance Naive Bayes classifier.	Improved 6% accuracy by comparing with the recommender using Naive Bayes only at Eclipse.	Additional cost to probabilistically label the unlabeled bug reports.
[47]	Machine-learning-based recommender	Utilized the feature selection algorithm to enhance the original Naive Bayes classifier by removing the noisy data.	Reduced the size of training set and improved the performance of original Naive Bayes classifier by up to 5% at the Eclipse project.	Needs more features.
[52]	Machine-learning-based recommender	Combined bug report-based analysis and develop-based analysis to develop an accurate developer recommender.	Improved the recall of Bugzie [45] and DREX [46] by 39.39 and 89.36%, respectively when recommending top-10 developers for the Gcc, OpenOffice, Mozilla, NetBeans and Eclipse projects.	Needs to extract more features to characterize the bug reports.

continued.

TABLE 8. continued.

Works	Mechanism	Method summary	Advantages	Disadvantages
[39]	Experience model-based recommender	Built a experience model to verify who has more experience on the new bug-fixing task.	Did not need to train the historical data.	Relatively lower precision (33.6% for top-1 recommendation) for Eclipse.
[45]	Experience model-based recommender	Utilized fuzzy set and cache-based techniques to model the developers' expertise for recommending the bug fixers.	Bugzie spent the less time (only 22 minutes) and achieved the higher accuracy (72%) for the Firefox, Eclipse, Apache, NetBeans, FreeDesktop, Gcc and Jazz projects.	Additional cost to build a list of technical terms extracted from the software systems.
[49]	Experience model-based recommender	Utilized location information of bugs, change history and expertise mapping to recommend the appropriate bug fixers.	Achieved satisfying accuracy by up to 81.44% when recommending top-3 developers for the AspectJ project.	Additional cost to locate the given bugs.
[36]	Tossing graph-based recommender	Reduced the tossing graph to improve the performance of developer recommendation.	Improved the precision accuracy of Naive Bayes and Naive Bayes Network-based recommender by up to 22.98 and 15.84%, respectively for Eclipse and Mozilla.	Additional cost to predict the path from the first assignee to the final assignee (real bug fixer).
[43]	Tossing graph-based recommender	Adopted multiple techniques such as refined classification, ranking function and multi-feature tossing graph to reduce the tossing path for improving the performance of fixer recommendation.	Reduced the length of tossing paths by up to 86% and achieved satisfying accuracy by up to 84% for Mozilla and 82.59% for Eclipse.	Needs more features to build multi-feature tossing graph.
[46]	Social network-based recommender	Utilized KNN with simple-frequency and social network metrics to execute automatic bug triage.	Demonstrated that both of two metrics-Outdegree and simple frequency achieved 60% recall when recommending top-10 developers in Mozilla Firefox project.	Additional cost to adjust parameters of the algorithm.
[29]	Social network-based recommender	Introduced a developer prioritization method via social network analysis to improve the fixer recommendation accuracy.	Improved the average prediction accuracy of SVM and Naive Bayes by 10 and 2%, respectively, at Mozilla and Eclipse.	Additional cost to analyze the developers' relationship.

continued.

TABLE 8. continued.

Works	Mechanism	Method summary	Advantages	Disadvantages
[50]	Topic model-based recommender	Developed DRETOM based on topic model to recommend the best developers for fixing new bugs.	Achieved higher recall up to 82 and 50% with top-5 and top-7 recommendations for Eclipse JDT and Mozilla Firefox, respectively, which performed better than SVM, KNN, out-degree and degree centrality based recommender.	Needs to adjust the parameters of topic model.
[51]	Topic model-based recommender	Adopted LDA to cluster the bug reports and created the activity profile for each developer based on history log and topic models, then used the activity profile to rank the developers for finding the appropriate bug fixers.	Achieved satisfying accuracy (average hit ratio of 88%) for the ATLAS Reconstruction, Eclipse BIRT and UNICASE projects.	Additional cost to adjust the parameters of LDA.
[30]	Topic model and social network-based recommender	Adopted topic model to find the similar bug reports with the new reported bug, then used multi-factors and social network techniques for finding the best fixers.	Improved prediction accuracy of DRETOM [50], activity file-based recommender [51] and Out-Degree [46] by 2–9, 5–11 and 2–10%, respectively, for Eclipse, Mozilla and NetBeans Java projects.	Additional cost to adjust the parameters of LDA and analyze the social network.
[53]	Topic model and social network-based recommender	Utilized LDA to extract the topics from historical bug reports and captured the developers' relations in the same topic based on the social network, then adopted the features extracted from these relations to rank the developers for finding the most appropriate fixers.	Improved *F*-measure of DRETOM [50] and Activity Profile [51] for Eclipse project by up to 3.3 and 16.5%, respectively.	Additional cost to adjust the parameters of LDA and analyze the social network.

approach combining *BUG* and *SOURCE* can successfully locate 27 bugs when recommending top-1 localization, which is much better than six bugs by only using *SOURCE*.

Zhou and his colleagues [59] proposed *BugLocator* to rank all source code files based on textual similarity between a new bug report and source code files, and combined the similarity between the new bug report and historical reports. By utilizing this hybrid similarity measure algorithm, Zhou *et al.* demonstrated that the *BugLocator* outperformed other bug localization methods using VSM, LDA, LSI and smoothed UM.

Kim *et al.* [60] proposed two-phase model to improve the performance of bug localization. In Phase 1, they utilized Naive Bayes to filter out the uninformative bug reports before predicting files to fix; in Phase 2, the prediction model accepted 'predictable' bug reports obtained from Phase 1 as the input to predict where a new bug should be fixed. The two-phase model can successfully predict buggy files to fix for 52–88% of all bug reports in eight modules of the Mozilla Firefox and Mozilla Core projects, with an average of 70%. The performance outperformed one-phase model that only used Naive Bayes to execute bug localization and *BugScout* [61].

4.1.2. *IR-based bug localization with combined information*

Nguyen *et al.* [61] proposed *BugScout*, a topic-based approach to locate the candidate buggy files for a new bug report. They utilized a topic model to represent the technique aspects of the software systems as topics, and correlated bug reports and corresponding buggy files via their shared topics. Thus *BugScout* can retrieve the correct buggy files where the given bug is. The evaluation results showed that *BugScout* can recommend buggy files correctly up to 45% of cases when recommending top-10 files.

Sisman and Kak [62] extracted the version histories of software project to estimate a prior probability distribution for bug proneness related to the buggy file in a given version of the project. Next, these prior knowledges are used in an IR framework to determine the posterior probability of a file being the location of a given bug. The evaluation results indicated that the proposed method can achieve $<80\%$ improvement in hit ratio for *BugScout* [61].

In the following study [63], Sisman and Kak proposed an automatic Query Reformulation (QR) to enrich a user's search query with certain specific additional terms extracted from the highest-ranked artifacts retrieved in response to the initial query. Then these additional terms were injected into an original query so that it can improve the performance of search engine for bug localization. In this work, they used the TF-IDF framework as a baseline [103] to evaluate the retrieval accuracy. The experimental results showed that the proposed QR approach can achieve a large accuracy improvement for original IR method by up to 66% for Eclipse and 90% for Chrome.

Saha *et al.* [64] developed *BLUiR* which is an IR-based open source toolkit to locate the given bugs. This toolkit not only uses the natural language information but also extracts the structure information of source code files, such as class and method names, to perform more accurate bug localization. Specifically, given a new bug report, the *summary* and *description* were extracted as two various document fields and source code files were parsed into four different document fields, including *class*, *method*, *variable* and *comments*. By introducing VSM (i.e., tf*idf model) to calculate the sum of the eight types of similarity values among these six kinds of document fields coming from the new bug report and the candidate source file, a ranked list of all candidate source files were generated so that the most appropriate location file can be found. The evaluation results showed that *BLUiR* performed better than *BugLocator*.

Wang and Lo [65] integrated version history, similar bug reports and structure information of source code to develop a new bug location tool called *AmaLgam* to retrieve relevant buggy files. They implemented *AmaLgam* on four open source projects, including AspectJ, Eclipse, SWT and ZXing to localize <3000 bugs. According to the evaluation results, compared with the approach proposed by Sisman and Kak [62], *AmaLgam* achieved 46.1% improvement in terms of mean average precision (MAP). For *BugLocator* [59], the improvement reached up to 24.4% in terms of MAP. Compared with *BLUiR* [64], *AmaLgam* achieved 16.4% improvement in terms of MAP.

Wang, Lo and Lawall [66] proposed a compositional VSMs for improving the performance of bug localization. This composite model $VSM_{composite}$ combined various VSM variants. VSM is represented by $tf * idf$. VSM variants are different combination types of tf and idf variants. Thereinto, tf includes five variants: $tf_n(t,d)$(i.e., natural term frequency), calculates the number of times that term t occurs in document d; $tf_l(t,d)$, takes the logarithm of the natural term frequency; $tf_L(t,d)$, normalizes $tf_l(t,d)$ by dividing it by the average logarithm of the other terms in the document d; $tf_a(t,d)$, normalizes $tf_n(t,d)$ by dividing it by the frequency of the term appearing the most times in document d; $tf_b(t,d)$(i.e., boolean term frequency), ignores the actual term frequency and only considers whether or not a term appears in a document d. idf contains three variants: $idf_n(t,D)$, gives the same weight for all terms in the document corpus D; $idf_l(t,D)$, is a standard inverse document frequency which computes the logarithm of the reciprocal of the document frequency; $idf_r(t,D)$, computes the logarithm of the ratio of documents not containing the term t and those containing the term t. Therefore, there are 15 VSM variants in the composite model $VSM_{\text{composite}}$. Given a bug report b and a candidate source file f, the similarity score between b and f is calculated by the composite model. By adopting this similarity measure, the potential location source files can be found. The evaluation results demonstrated that $VSM_{\text{composite}}$ outperformed VSM with the standard tf-idf weighting schema.

Note that combining structure information of source code files or utilizing the compositional VSMs can help to improve the performance of bug localization. However, the running cost and algorithm complexity are also increased.

4.1.3. *Comparison of bug localization approaches*

We provide the comparison results of bug localization approaches in Table 9. Note that these representative studies also adopted IR-based bug localization method to search the source files where the reported bug is. By considering the additional information such as structure information [64] or combining more VSM variants [66], the accuracy of bug location recommendation has shown to improve.

4.2. Survey on automatic patch (repair) generation approaches

After assignees fix the given bugs, some software patches (repairs) will be generated in the fixing process. Unfortunately, the process is tedious and requires intensive human resources to generate patches. To address this problem, it is necessary to develop the automatic patch generation approach. However, automatic patch generation faces two challenges. Firstly, we must extract the fix patterns from human-written patches or bug reports; secondly, we need to verify the correctness of

TABLE 9. Comparison of automatic bug localization approaches.

Works	Mechanism	Method summary	Advantages	Disadvantages
[54, 55]	IR-based bug localization	Used LDA to execute the source files retrieval.	Only needs to extract the textual information of given bug report and source files.	Needs to adjust the number of topics K for different data sets such as Mozilla and Eclipse.
[56]	IR-based bug localization	Evaluated the different performance of bug localization by using the different IR models such as smoothed UM, SVM, LDA, LSA and CBDM.	Demonstrated that smooth UM and VSM achieved the MAP value by up to 14.54 and 7.96%, respectively, which performed better than others at the AspectJ project.	Needs to adjust the parameters of appropriate IR models for different data set.
[57]	IR-based bug localization	Combined the best-performing classifiers by adopting their best configurations to search the buggy file.	Achieved the improvements ranged from −2 to 12% according to different projects (i.e., Eclipse, Mozilla and Jazz) and combination methods used.	Additional cost to evaluate the best configurations of each classifier.
[58]	IR-based bug localization	Combined the similarity between bug reports and the similarity between the new bug report and the source code to execute the bug localization.	Successfully located 27 bugs when recommending top-1 buggy file at ArgoUML, JabRef, jEdit and muCommander, which is much better than six bugs by only measuring the similarity between the new bug report and the source code.	
[59]	IR-based bug localization	Computed textual similarity between a new bug report and source files, and combined the similarity between the new report and historical bug reports to find the correct location file.	Successfully located <30, 50 and 60% bugs when recommending top-1, top-5 and top-10 buggy files, which performed better than VSM, LDA, LSI and SUM for Eclipse, SWT, AspectJ and ZXing projects.	Additional cost to adjust the weighting vector for different data set.
[60]	IR-based bug localization	Utilized Naive Bayes to filter out the uninformative bug reports and predicted the buggy files of the given bug.	Successfully predicted files to fix for 52–88% of all bug reports in eight modules of Mozilla Firefox and Mozilla Core projects, which outperformed one-phase model-based localization approach that only used Naive Bayes and *BugScout* [61].	Additional cost to filter the uninformative bug reports.

continued.

TABLE 9. continued.

Works	Mechanism	Method summary	Advantages	Disadvantages
[61]	IR-based bug localization with technique aspects	Utilized topic model to retrieve the buggy files belonging to the same topics related to technique aspects with the given bug.	Reduced the search space.	Relatively lower accuracy (45% of bugs were located) for Jazz, Eclipse, AspectJ and ArgoUML.
[62]	IR-based bug localization with version history	Utilized version history to enhance the performance of bug localization.	Achieve <80% improvement in hit ratio for *BugScout* [61] at the AspectJ project.	Needs to adjust the parameters.
[63]	IR-based bug localization with Query Reformulation	Adopted Query Reformulation (QR) to enrich a user's search query for improving the performance of bug localization.	Achieved the accuracy improvement for original IR method by up to 66% for Eclipse and 90% for Chrome.	Needs to adjust the parameters.
[64]	IR-based bug localization with structure information	Combined the natural language and structure information to compute the similarity between a new bug report and source files for locating the bugs.	Relatively higher accuracy (60% for AspectJ and 63% for Eclipse) than BugLocator [59] (56% for AspectJ and Eclipse) and *BugScout*[61] (35% for AspectJ and 31% for Eclipse) when recommending top-10 buggy files.	Needs to extract and parse the structure information.
[65]	IR-based bug localization with hybrid information	Integrated version history, similar bug reports and structure information to retrieve relevant buggy files.	Improved the accuracy of [59, 62, 64] by 46.1%, 24.4%, and 16.4%, respectively, for the bug reports from AspectJ, Eclipse and SWT.	Additional cost to extract and pre-process the version history and structure information.
[66]	IR-based bug localization with the composite model	Utilized the composite model combining 15 VSM variants to compute the similarity between a new bug report and source files to locate the buggy files.	Improved hit ratio at top-5 buggy files of standard VSM by 18.4% for the AspectJ, Eclipse, SWT and ZXing projects.	Increased the algorithm complexity.

generated patches. To overcome these difficulties, the textual parsing and machine-learning-based classification techniques are introduced. The related automatic patch generation methods [67–71] have been proposed to replace the manual patch generation.

4.2.1. *GP-based patch (repair) generation*

As an early research, Weimer *et al.* introduced Genetic programming (GP) [67], which is a computational method in biological evolution theoretical system, to maintain variants of the software program by using crossover operators and mutation operators such as statement addition, replacement and removal. Given the test cases, the proposed approach can evaluate each variant until one of the variants passes all test cases. This variant passing all test cases was regarded as a successful patch. Authors reported the experimental results demonstrating that the proposed method can generate patches for 10 different C programs with the average success rate of 54%. However, this GP-based patch generation approach relies on random program mutations so that it may generate nonsensical patches.

In the following work, Goues *et al.* [68] extended their previous work [67] to propose GenProg, an automated method for generating the repairs (i.e., patches) for real-world bugs. GenProg adopted an extended form of GP to evolve a program variant that retains required functionality but is not susceptible to a given bug. GenProg used the input test cases to evaluate the fitness, and the variants with the high fitness are selected for continued evolution. The GP process is successful when the variant passes all tests encoding the required behavior and does not fail those encoding the bug. Experimental results showed that GenProg can successfully generate the patches in 16 C programs.

In [69], Goues *et al.* proposed an algorithm improvement for their prior work GenProg [68]. This new approach used off-the-shelf cloud computing as a framework for exploiting search-space parallelism as well as a source of grounded cost measurements. The experimental results showed that the new approach can find 68% more patches than previous work [67].

4.2.2. Pattern-based patch (repair) generation

In order to avoid the problem existing in GP-based patch generation, Kim and his group [70] proposed a pattern-based automatic patch generation method (i.e., PAR). By mining the common patches using *groums* (A groum is a graph-based model for representing object usage) from human-written patches, they got six patterns to create 10 fix templates, which are automatic program editing scripts. Specifically, PAR first extracted the bugs from an open source project and identified bug locations. Then, it generated program variants by using fix templates. These templates modified the source code that appears in the source files. The variants were treated as the patch candidates, and were evaluated by using test cases. The patch candidate that passed all test cases is verified as the correct patch of the given bug. The evaluation results showed that PAR produced more acceptable patches than GP-based patch generation approach [67].

4.2.3. Bug report analysis-based patch (repair) generation

Different from the patch generation methods proposed in [67–71], Liu and his colleagues [71] proposed a bug-fixing patch generation approach-*R2Fix* to automatically produce the patches based on bug reports without test cases and specifications. *R2Fix* contains three components: *Classifier*, *Pattern Parameter Extractor* and P*atch Generator*. *Classifier* is responsible for classifying bug reports into the target bug types after shorting and parsing them; *Pattern Parameter Extractor* is used to analyze the candidate bug reports and source code to extract pattern parameters (e.g., pointer names, buffer lengths, etc.); *Patch Generator* uses the pattern parameters, the fix patterns for each target bug type, and the source code repository to generate patches automatically. The experimental results showed that *R2Fix* can generate the 57 correct patches and saved the time of bug diagnosis and patch generation time.

4.2.4. Comparison of automatic patch (repair) generation approaches

In Table 10, we summarize the comparison results of three different automatic patch generation approaches [67–71]. Note that the studies [67–70] need to utilize test cases to evaluate whether a candidate patch is correct or not. However, Liu *et al.* proposed an approach in [71] to generate the patches based on bug report analysis without test cases. This approach can generate more correct patches and save the patch generation time.

5. DISCUSSION

5.1. Bug resolution using multi-techniques

By investigating the research papers on each task in bug resolution, we find that various knowledge and techniques coming from non-SE area were utilized to resolve the problems that exist in software maintenance. Table 11 summarizes the techniques adopted in each task of bug resolution, respectively.

In this table, we note that most software phases except patch generation adopted NLP to pre-process the bug reports and source code files. Patch generation is a special case because the proposed methods do not need to utilize the extracted terms from documents to execute IR-based approaches, including VSM, BM25 and TM. Moreover, other popular techniques like machine learning and VSM were employed in most of phases. Through our investigation and analysis, machine-learning techniques (e.g., SVM, KNN) are useful for verifying which sentences may form the summary of a new bug report, predicting the factors, recommending the fixers and generating the patches, and VSM is a simple and effective method to calculate the similarity measure between two documents. Social network was introduced to analyze the relationship between developers so it can only be used to execute bug triage. As another similarity measure, BM25 that includes $BM25F$ and $BM25_{ext}$ has been shown to have the relatively higher accuracy than VSM so that it has been widely employed in factor prediction and duplicates detection. Topic model is one of IR model, which is a way to categorize the documents so that it can be adopted to execute the severity prediction, duplicates detection, bug triage and bug localization tasks.

5.2. Which elements affect the performance of bug resolution?

According to the investigation and analysis of previous studies using multi-techniques on bug understanding, bug triage and bug fixing, we found that some elements can impact the performance of bug resolution. These elements are summarized as follows.

(1) *Data element*: Additional non-textual information such as execution information and structure information can improve the performance of bug resolution. For example, execution information increased the accuracy of duplicates detection [9], and structure information improved the accuracy of bug localization [64]. Moreover, some non-textual factors of bug reports such as product, component, priority and severity can enhance the capability of bug

TABLE 10. Comparison of automatic patch (repair) generation approaches.

Works	Mechanism	Method summary	Advantages	Disadvantages
[67]	GP-based patch (repair) generation	Utilized test cases to evaluate program variants maintained by GP until a variant passing all test cases. This variant was regarded as the correct one.	Does not require specifications, program annotations or special coding practices.	Random program mutations may lead to nonsensical patches.
[68]	GP-based patch (repair) generation	Adopted an extended form of GP to implement automatic patch generation.	Successfully generated the patches in 16 C programs such as gcd, zune, etc.	Non-deterministic properties of test cases may affect the generation results.
[69]	GP-based patch (repair) generation	Used cloud computing as a framework for improving the performance of GenProg introduced in [68] by exploiting its search-space parallelism.	Generated 68% more patches than previous work [67] for eight open source projects, including fbc, gmp, gzip, libtiff, lighttpd, php, python and wireshark.	Non-deterministic properties of test cases may affect the generation results.
[70]	Fix pattern-based patch (repair) generation	Utilized the fix patterns mining from human-written patches to generate the candidate patches, the patch passing all test cases was regarded as the correct one.	Successfully generated patches for 27 of 119 bugs, which is better than GenProg [67, 68] for only 16 bugs in Mozilla, Eclipse and Apache projects.	Needs test cases to evaluate the candidate patches.
[71]	Bug report analysis-based patch (repair) generation	Combined past fix patterns, machine- learning techniques and semantic patch generation techniques to automatically generate the patches at three projects, including the Linux kernel, Mozilla and Apache.	Does not require any specifications or test cases, and can save the patch generation time.	Increased the cost of algorithm running.

resolution. For example, multi-factors without priority were used to predict the priority of bug reports and showed the acceptable prediction accuracy (The F-measure achieved a relative improvement of 58.61% than SVM-MultiClass) [22], and the multiple factors were adopted to increase the accuracy of duplicates detection [11].

(2) *Human element*: In the whole bug fixing activity, developers constitute a social network which influences the process of bug understanding, bug traige and bug fixing. Therefore, social network-based analysis is very useful for facilitating the tasks in three phases. For instance in [46], by capturing developers' behavior (i.e., commenting activities) on bug fixing, a series of social network metrics like out-degree were utilized to execute the task of assignee recommendation and showed a better accuracy.

(3) *Technique element*: An effective technique generally can help to improve the performance of predictive tasks. For example, topic model can improve the accuracy of duplicates detection [12]; and it was also utilized to increase the accuracy of assignee recommendation [50]. Using more-accurate similarity measures like $BM25Fext$ [28] or adopting the variants of VSM (e.g., 15 different VSMs in [66]), the accuracy of IR-based bug resolution approaches is improved.

Even if these good practices can help us improve the accuracy of automatic bug resolution, employing them needs additional cost to execute the corresponding algorithms. Therefore, it is necessary to find the trade-off between the performance and cost.

TABLE 11. Automatic bug resolution using multi-techniques.

	Technique					
Phase	NLP[a]	ML[b]	SN[c]	VSM[d]	BM25[e]	TM[f]
Summarization	[3–6]	[3–5]	N	[5, 6]	N	N
Duplicates detection	[7–17]	[8, 10, 13, 16]	N	[7–10]	[11–13, 16, 17]	[12, 14]
Priority prediction	[18–22]	[18–22]	N	[19–21]	[22]	N
Severity prediction	[23–26, 28–30]	[23–26, 28–30]	N	[24, 25, 29]	[28]	[30]
Reopened status prediction	[33]	[29, 31–33]	[29]	[33]	N	N
Blocking status prediction	N	[34, 35]	N	N	N	N
Bug triage	[36–44] [29, 30, 46–48, 50–53]	[36, 37, 37, 38, 40–43] [29, 44, 46–48, 52]	[30, 53] [29, 46]	[39–43] [29, 46–48]	N	[30, 50] [52, 53]
Bug localization	[54–66]	[60]	N	[56–59, 62–66]	N	[54–57, 61]
Patch generation	N	[71]	N	N	N	N

[a]Natural language processing.
[b]Machine learning.
[c]Social network.
[d]VSM and variants.
[e]BM25 and variants.
[f]Topic model.

6. FUTURE RESEARCH DIRECTIONS FOR BUG RESOLUTION

In the future, some new research directions on bug resolution may be further studied. We describe them according to the different phases of bug resolution, including bug understanding, bug triage and bug fixing.

6.1. Bug understanding

In the phase of bug understanding, the previous studies focus on bug summarization, duplicates detection and feature prediction. We found that most of the proposed approaches utilized traditional machine-learning algorithms such as Naive Bayes, SVM and KNN to execute the above-mentioned three tasks. In the future, some more effective machine-learning algorithms such as deep learning [104] may be adopted to implement these tasks. Deep learning has already been widely implemented in image and text processing [105]. Different from traditional machine-learning algorithms such as SVM, it models high-level abstractions in data using model architectures which may improve the performance of related tasks based on natural language processing [106] in the bug understanding phase.

The quality of bug reports [107, 108] is a very important factor to affect the performance of the tasks on bug understanding. Obviously, a bug report that includes more noisy data and lacks the detailed description of the related bug cannot help developers complete the tasks in the phase of bug understanding. Thus, enhancing bug report is a necessary step to improve the quality of bug reports by removing the noisy sentences and adding the detailed bug description so that it can improve the results of bug understanding.

6.2. Bug triage

The purpose of bug triage is recommending the most appropriate developers to fix the given bugs. The previous studies devoted to utilize the historical bug reports to develop the automatic recommender. Actually, in real open source and commercial projects, other types of communication such as email and instant messaging are also used to discuss how to fix the given bugs [109]. Thus, except for bug reports, other software artifacts such as emails from other data resources (e.g., email repositories) can provide further knowledge to help us develop more effective algorithms for performing bug triage. For example, the information showed in emails can help to analyze the developers' activities [110] so that extracting the email data may enhance the developer recommender.

6.3. Bug fixing

Bug localization and patch generation are the two major tasks in bug fixing. The previous studies developed a series of automatic tools for locating the bugs and producing the corresponding patches. These works mainly served for traditional desktop software. Recent years, with increasing number of mobile applications (apps), resolving the bugs appearing in apps becomes an important and challenging research direction [111, 112]. Due to the different characteristics (e.g., little spread of historical data in each app) with traditional desktop software, the previous approaches for bug localization and patch generation in desktop software may not be used in mobile apps. Thus the new bug fixing methods for apps need to be developed in the future.

In the tasks of bug localization and patch generation, code analysis plays an important role. For example, in the previous IR-based bug localization approaches, in order to find the correct buggy file, the general way is to compute the textual similarity between the bug report and the source code. The source code files are the structured documents [64], thus, some code analysis techniques such as graph-based analysis approaches [113] and graph-based similarity measures [114, 115] can further provide more detailed information and features so that they can be utilized for improve the performance of bug localization and patch generation in the future.

7. CONCLUSION

In this paper, we survey the previous studies on bug understanding, bug triage and bug fixing, and show their advantages and disadvantages. Moreover, we summarize multi-techniques which were utilized to implement each task in three phases of bug resolution, and present the factors which affect the performance of these tasks. Finally, we introduce the future research directions on bug resolution. We expect that this complete survey article can contribute to both scholars and developers who work on the field of software maintenance.

FUNDING

This work is supported in part by the Hong Kong GRF (No.PolyU5389/13E), the National Natural Science Foundation of China (No.61202396), the HKPolyU Research Grant (G-UA3X), and the Hong Kong ITF(No. UIM/285).

REFERENCES

[1] Charette, R. (2005) Why software fails [software failure]. *IEEE Spectrum*, **42**, 42–49.

[2] Erlikh, L. (2000) Leveraging legacy system dollars for e-busi-ness. *IT Professional*, **2**, 17–23.

[3] Rastkar, S., Murphy, G.C. and Murray, G. (2010) Summarizing Software Artifacts: A Case Study of Bug Reports. *Proceedings of ICSE 10, Cape Town*, South Africa, May 2–8, pp. 505–514. ACM, New York.

[4] Rastkar, S., Murphy, G.C. and Murray, G. (2014) Automatic summarization of bug reports. *IEEE Transactions on Software Engineering*, **40**, 366–380.

[5] Mani, S., Catherine, R., Sinha, V.S. and Dubey, A. (2012) Ausum: Approach for Unsupervised Bug Report Summarization. *Proceedings of FSE 12, Cary*, North Carolina, November 11–16, pp. 11:1–11:11. ACM, New York.

[6] Lotufo, R., Malik, Z. and Czarnecki, K. (2012) Modelling the 'Hurried' Bug Report Reading Process to Summarize Bug Reports. *Proceedings of ICSM* 12, Trento, Italy, September 23–30, pp. 430–439. IEEE Computer Society, Los Alamitos, CA.

[7] Runeson, P., Alexandersson, M. and Nyholm, O. (2007) Detection of Duplicate Defect Reports Using Natural Language Processing. *Proceedings of ICSE 07*, Minneapolis, MN, May 20–26, pp. 499–510. IEEE Computer Society, Los Alamitos, CA.

[8] Jalbert, N. and Weimer, W. (2008) Automated Duplicate Detection for Bug Tracking Systems. *Proceedings of DSN 08*, Anchorage, Alaska, USA, June 24–27, pp. 52–61. IEEE Computer Society, Los Alamitos, CA.

[9] Wang, X., Zhang, L., Xie, T., Anvik, J. and Sun, J. (2008) An Approach to Detecting Duplicate Bug Reports Using Natural Language and Execution Information. *Proceedings of ICSE 08*, Leipzig, Germany, May 10–18, pp. 461–470. ACM, New York.

[10] Sun, C., Lo, D., Wang, X., Jiang, J. and Khoo, S.-C. (2010) A Discriminative Model Approach for Accurate Duplicate Bug Report Retrieval. *Proceedings of ICSE 10*, Cape Town, South Africa, May 1–8, pp. 45–54. ACM, New York.

[11] Sun, C., Lo, D., Khoo, S.-C. and Jiang, J. (2011) Towards More Accurate Retrieval of Duplicate Bug Reports. *Proceedings of ASE* 11, Lawrence, KS, November 6–10, pp. 253–262. IEEE Computer Society, Los Alamitos, CA.

[12] Nguyen, A.T., Nguyen, T.T., Nguyen, T.N., Lo, D. and Sun, C. (2012) Duplicate Bug Report Detection with a Combination of Information Retrieval and Topic Modeling. *Proceedings of ASE* 12, Essen, Germany, September 3–7, pp. 70–79. ACM, New York.

[13] Alipour, A., Hindle, A. and Stroulia, E. (2013) A Contextual Approach Towards More Accurate Duplicate Bug Report Detection. *Proceedings of MSR* 13, San Francisco, California, May 18-19, pp. 183–192. IEEE, Piscataway, NJ.

[14] Hindle, A., Alipour, A. and Stroulia, E. (2015) A contextual approach towards more accurate duplicate bug report detection and ranking. *Empirical Software Engineering*, **Online**, 1–43.

[15] Sureka, A. and Jalote, P. (2010) Detecting Duplicate Bug Report Using Character n-Gram-Based Features. *Proceedings of APSEC 10*, Sydney, Australia, November 30–December 3, pp. 366–374. IEEE Computer Society, Los Alamitos, CA.

[16] Tian, Y., Sun, C. and Lo, D. (2012) Improved Duplicate Bug Report Identification. *Proceedings of CSMR* 12, Szeged, Hungary, March 27–30, pp. 385–390. IEEE Computer Society, Los Alamitos, CA.

[17] Aggarwal, K., Rutgers, T., Timbers, F., Hindle, A., Greiner, R. and Stroulia, E. (2015) Detecting Duplicate Bug Reports with Software Engineering Domain Knowledge. *Proceedings of SANER* 15, Montreal, QC, Canada, March 2–6, pp. 211–220. IEEE, Piscataway, NJ.

[18] Yu, L., Tsai, W.-T., Zhao, W. and Wu, F. (2010) Predicting Defect Priority based on Neural Networks. *Proceedings of ADMA* 10, Chongqing, China, November 19–21, pp. 356–367. Springer, Berlin.

[19] Kanwal, J. and Maqbool, O. (2012) Bug prioritization to facilitate bug report triage. *Journal of Computer Science and Technology*, **27**, 397–412.

[20] Sharma, M., Bedi, P., Chaturvedi, K. and Singh, V. (2012) Predicting the Priority of a Reported Bug using Machine Learning Techniques and Cross Project Validation. *Proceedings of ISDA* 12, Kochi, India, November 27–29, pp. 539–545. IEEE, Piscataway, NJ.

[21] Alenezi, M. and Banitaan, S. (2013) Bug Reports Prioritization: Which Features and Classifier to Use?. *Proceedings of ICMLA* 13, Miami, FL, December 4–7, pp. 112–116. IEEE Computer Society, Los Alamitos, CA.

[22] Tian, Y., Lo, D. and Sun, C. (2013) Drone: Predicting Priority of Reported Bugs by Multi-factor Analysis. *Proceedings of ICSM* 13, Eindhoven, The Netherlands, September 22–28, pp. 200–209. IEEE Computer Society, Los Alamitos, CA.

[23] Lamkanfi, A., Demeyer, S., Giger, E. and Goethals, B. (2010) Predicting the Severity of a Reported Bug. *Proceedings of MSR* 10, Cape Town, South Africa, May 2–3, pp. 1–10. IEEE, Piscataway, NJ.

[24] Lamkanfi, A., Demeyer, S., Soetens, Q.D. and Verdonck, T. (2011) Comparing Mining Algorithms for Predicting the Severity of a Reported Bug. *Proceedings of CSMR* 11, Oldenburg, Germany, March 1–4, pp. 249–258. IEEE Computer Society, Los Alamitos, CA.

[25] Menzies, T. and Marcus, A. (2008) Automated Severity Assessment of Software Defect Reports. *Proceedings of ICSM* 08, Beijing, China, September 28–October 4, pp. 346–355. IEEE, Piscataway, NJ.

[26] Yang, C.-Z., Hou, C.-C., Kao, W.-C. and Chen, X. (2012) An Empirical Study on Improving Severity Prediction of Defect Reports Using Feature Selection. *Proceedings of APSEC* 12, Hong Kong, China, December 4–7, pp. 240–249. IEEE, Piscataway, NJ.

[27] Bhattacharya, P., Iliofotou, M., Neamtiu, I. and Faloutsos, M. (2012) Graph-Based Analysis and Prediction for Software Evolution. *Proceedings of ICSE* 12, Zurich, Switzerland, June 2–9, pp. 419–429. IEEE, Piscataway, NJ.

[28] Tian, Y., Lo, D. and Sun, C. (2012) Information Retrieval Based Nearest Neighbor Classification for Fine-Grained Bug Severity Prediction. *Proceedings of WCRE* 12, Kingston, ON, Canada, October 15–18, pp. 215–224. IEEE Computer Society, Los Alamitos, CA.

[29] Xuan, J., Jiang, H., Ren, Z. and Zou, W. (2012) Developer Prioritization in Bug Repositories. *Proceedings of ICSE* 12, Zurich, Switzerland, June 2–9, pp. 25–35. IEEE, Piscataway, NJ.

[30] Yang, G., Zhang, T. and Lee, B. (2014) Towards Semi-automatic Bug Triage and Severity Prediction Based on Topic Model and Multi-feature of Bug Reports. *Proceedings of COMPSAC* 14, Vasteras, Sweden, July 21–25, pp. 97–106. IEEE, Piscataway, NJ.

[31] Shihab, E., Ihara, A., Kamei, Y., Ibrahim, W.M., Ohira, M., Adams, B., Hassan, A.E. and Matsumoto, K.-i. (2010) Predicting Re-opened Bugs: A Case Study on the Eclipse Project. *Proceedings of WCRE* 10, Beverly, MA, October 13–16, pp. 249–258. IEEE Computer Society, Los Alamitos, CA.

[32] Xia, X., Lo, D., Wang, X., Yang, X., Li, S. and Sun, J. (2013) A Comparative Study of Supervised Learning Algorithms for Re-opened Bug Prediction. *Proceedings of CSMR* 13, Genova, Italy, March 5–8, pp. 331–334. IEEE Computer Society, Los Alamitos, CA.

[33] Xia, X., Lo, D., Shihab, E., Wang, X. and Zhou, B. (2014) Automatic, high accuracy prediction of reopened bugs. *Automated Software Engineering*, **22**, 75–109.

[34] ValdiviaGarcia, H. and Shihab, E. (2014) Characterizing and Predicting Blocking Bugs in Open Source Projects. *Proceedings of MSR* 14, Hyderabad, India, May 31–June 1, pp. 72–81. ACM, New York.

[35] Xia, X., Lo, D., Shihab, E., Wang, X. and Yang, X. (2015) Elblocker: predicting blocking bugs with ensemble imbalance learning. *Information and Software Technology*, **61**, 93–106.

[36] Jeong, G., Kim, S. and Zimmermann, T. (2009) Improving Bug Triage with Bug Tossing Graphs. *Proceedings of ESEC/FSE* 09, Amsterdam, The Netherlands, August 24–28, pp. 111–120. ACM, New York.

[37] Čubranić, D. and Murphy, G.C. (2004) Automatic Bug Triage Using Text Categorization. *Proceedings of SEKE* 04, Banff, Alberta, Canada, June 20–24, pp. 92–97. Knowledge Systems Institute Graduate School, Skokie.

[38] Anvik, J., Hiew, L. and Murphy, G.C. (2006) Who should fix this bug?. *Proceedings of ICSE* 06, Shanghai, China, May 20–28, pp. 361–370. ACM, New York.

[39] Matter, D., Kuhn, A. and Nierstrasz, O. (2009) Assigning Bug Reports Using a Vocabulary-Based Expertise Model of Developers. *Proceedings of MSR* 09, Vancouver, BC, May 16–17, pp. 131–140. IEEE, Piscataway, NJ.

[40] Lin, Z., Shu, F., Yang, Y., Hu, C. and Wang, Q. (2009) An Empirical Study on Bug Assignment Automation Using Chinese Bug Data. *Proceedings of ESEM* 09, Lake Buena Vista, Florida, October 15–16, pp. 451–455. IEEE, Piscataway, NJ.

[41] Ahsan, S.N., Ferzund, J. and Wotawa, F. (2009) Automatic Software Bug Triage System (bts) based on Latent Semantic Indexing and Support Vector Machine. *Proceedings of ICSEA* 09, Porto, Portugal, September 20–25, pp. 216–221. IEEE Computer Society, Los Alamitos, CA.

[42] Xuan, J., Jiang, H., Ren, Z., Yan, J. and Luo, Z. (2010) Automatic Bug Triage using Semi-Supervised Text Classification. *Proceedings of SEKE* 10, Redwood City, San Francisco Bay, CA, July 1–3, pp. 209–214. Knowledge Systems Institute Graduate School, Skokie, IL.

[43] Bhattacharya, P. and Neamtiu, I. (2010) Fine-Grained Incremental Learning and Multi-feature Tossing Graphs to Improve Bug Triaging. *Proceedings of ICSM* 10, Timisoara, Romania, September 12–18, pp. 1–10. IEEE Computer Society, Los Alamitos, CA.

[44] Anvik, J. and Murphy, G.C. (2011) Reducing the effort of bug report triage: recommenders for development-oriented decisions. *ACM Transactions on Software Engineering and Methodology*, **20**, 10:1–10:35.

[45] Tamrawi, A., Nguyen, T.T., Al-Kofahi, J.M. and Nguyen, T.N. (2011) Fuzzy Set and Cache-Based Approach for Bug Triaging. *Proceedings of ESEC/FSE* 11, Szeged, Hungary, September 5–9, pp. 365–375. ACM, New York.

[46] Wu, W., Zhang, W., Yang, Y. and Wang, Q. (2011) Drex: Developer Recommendation with k-Nearest-Neighbor Search and Expertise Ranking. *Proceedings of APSEC* 11, Ho Chi Minh, Vietnam, December 5–8, pp. 389–396. IEEE Computer Society, Los Alamitos, CA.

[47] Zou, W., Hu, Y., Xuan, J. and Jiang, H. (2011) Towards Training Set Reduction for Bug Triage. *Proceedings of COMPSAC* 11,

Munich, Germany, July 18–22, pp. 576–581. IEEE Computer Society, Los Alamitos, CA.

[48] Bhattacharya, P., Neamtiu, I. and Shelton, C.R. (2012) Automated, highly-accurate, bug assignment using machine learning and tossing graphs. *Journal of Systems and Software*, **85**, 2275–2292.

[49] Servant, F. and Jones, J. *et al.* (2012) Whosefault: Automatic Developer-to-Fault Assignment through Fault Localization. *Proceedings of ICSE* 12, Zurich, Switzerland, June 2–9, pp. 36–46. IEEE, Piscataway, NJ.

[50] Xie, X., Zhang, W., Yang, Y. and Wang, Q. (2012) Dretom: Developer Recommendation Based on Topic Models for Bug Resolution. *Proceedings of PROMISE* 12, Lund, Sweden, September 21–22, pp. 19–28. ACM, New York.

[51] Naguib, H., Narayan, N., Brugge, B. and Helal, D. (2013) Bug Report Assignee Recommendation Using Activity Profiles. *Proceedings of MSR* 13, San Francisco, CA, May 18–19, pp. 22–30. IEEE Computer Society, Los Alamitos, CA.

[52] Xia, X., Lo, D., Wang, X. and Zhou, B. (2013) Accurate Developer Recommendation for Bug Resolution. *Proceedings of WCRE* 13, Koblenz, Germany, October 14–17, pp. 72–81. IEEE Computer Society, Los Alamitos, CA.

[53] Zhang, T., Yang, G., Lee, B. and Lua, E.K. (2014) A Novel Developer Ranking Algorithm for Automatic Bug Triage Using Topic Model and Developer Relations. *Proceedings of APSEC* 14, Jeju, South Korea, December 1–4, pp. 223–230. IEEE, Piscataway, NJ.

[54] Lukins, S., Kraft, N. and Etzkorn, L. (2008) Source Code Retrieval for Bug Localization Using Latent Dirichlet Allocation. *Proceedings of WCRE* 08, Antwerp, Belgium, October 15–18, pp. 155–164. IEEE Computer Society, Los Alamitos, CA.

[55] Lukins, S.K., Kraft, N.A. and Etzkorn, L.H. (2010) Bug localization using latent Dirichlet allocation. *Information and Software Technology*, **52**, 972–990.

[56] Rao, S. and Kak, A. (2011) Retrieval from Software Libraries for Bug Localization: A Comparative Study of Generic and Composite Text Models. *Proceedings of MSR* 11, Waikiki, Honolulu, HI, May 21–22, pp. 43–52. ACM, New York.

[57] Thomas, S.W., Nagappan, M., Blostein, D. and Hassan, A.E. (2013) The impact of classifier configuration and classifier combination on bug localization. *IEEE Transactions on Software Engineering*, **39**, 1427–1443.

[58] Davies, S., Roper, M. and Wood, M. (2012) Using Bug Report Similarity to Enhance Bug Localisation. *Proceedings of WCRE* 12, Kingston, ON, Canada, October 15–18, pp. 125–134. IEEE Computer Society, Los Alamitos, CA.

[59] Zhou, J., Zhang, H. and Lo, D. (2012) Where Should the Bugs be Fixed? - More Accurate Information Retrieval-Based Bug Localization Based on Bug Reports. *Proceedings of ICSE* 12, Zurich, Switzerland, June 2–9, pp. 14–24. IEEE, Piscataway, NJ.

[60] Kim, D., Tao, Y., Kim, S. and Zeller, A. (2013) Where should we fix this bug? A two-phase recommendation model. *IEEE Transactions on Software Engineering*, **39**, 1597–1610.

[61] Nguyen, A.T., Nguyen, T.T., Al-Kofahi, J., Nguyen, H.V. and Nguyen, T.N. (2011) A Topic-Based Approach for Narrowing the Search Space of Buggy Files from a Bug Report. *Proceedings of ASE* 11, Lawrence, KS, November 6–10, pp. 263–272. IEEE Computer Society, Los Alamitos, CA.

[62] Sisman, B. and Kak, A.C. (2012) Incorporating Version Histories in Information Retrieval Based Bug Localization. *Proceedings of MSR* 12, Zurich, Switzerland, June 2–3, pp. 50–59. IEEE Computer Society, Los Alamitos, CA.

[63] Sisman, B. and Kak, A.C. (2013) Assisting Code Search with Automatic Query Reformulation for Bug Localization. *Proceedings of MSR* 13, San Francisco, CA, May 18-19, pp. 309–318. IEEE Computer Society, Los Alamitos, CA.

[64] Saha, R., Lease, M., Khurshid, S. and Perry, D. (2013) Improving Bug Localization Using Structured Information Retrieval. *Proceedings of ASE* 13, Silicon Valley, CA, November 11–15, pp. 345–355. IEEE, Piscataway, NJ.

[65] Wang, S. and Lo, D. (2014) Version History, Similar Report, and Structure: Putting Them Together for Improved Bug Localization. *Proceedings of ICPC* 14, Hyderabad, India, June 2–3, pp. 53–63. ACM, New York.

[66] Wang, S., Lo, D. and Lawall, J. (2014) Compositional Vector Space Models for Improved Bug Localization. *Proceedings of ICSME* 14, Victoria, BC, Canada, September 29–October 3, pp. 171–180. IEEE Computer Society, Los Alamitos, CA.

[67] Weimer, W., Nguyen, T., Le Goues, C. and Forrest, S. (2009) Automatically Finding Patches Using Genetic Programming. *Proceedings of ICSE* 09, Vancouver, Canada, May 16–24, pp. 364–374. IEEE Computer Society, Los Alamitos, CA.

[68] Goues, C.L., Nguyen, T., Forrest, S. and Weimer, W. (2012) Genprog: a generic method for automatic software repair. *IEEE Transactions on Software Engineering*, **38**, 54–72.

[69] Le Goues, C., Dewey-Vogt, M., Forrest, S. and Weimer, W. (2012) A Systematic Study of Automated Program Repair: Fixing 55 out of 105 Bugs for $8 Each. *Proceedings of ICSE* 12, Zurich, Switzerland, June 2–9, pp. 3–13. IEEE, Piscataway, NJ.

[70] Kim, D., Nam, J., Song, J. and Kim, S. (2013) Automatic Patch Generation Learned from Human-Written Patches. *Proceedings of ICSE* 13, San Francisco, CA, May 18–26, pp. 802–811. IEEE, Piscataway, NJ.

[71] Liu, C., Yang, J., Tan, L. and Hafiz, M. (2013) R2fix: Automatically Generating Bug Fixes from Bug Reports. *Proceedings of ICST* 13, Luxembourg, March 18–22, pp. 282–291. IEEE Computer Society, Los Alamitos, CA.

[72] Strate, J. and Laplante, P. (2013) A literature review of research in software defect reporting. *IEEE Transactions on Reliability*, **62**, 444–454.

[73] Zhang, J., Wang, X., Hao, D., Xie, B., Zhang, L. and Mei, H. (2015) A survey on bug-report analysis. *Science China Information Sciences*, **58**, 1–24.

[74] Dietterich, T.G. (1998) Approximate statistical tests for comparing supervised classification learning algorithms. *Neural Computation*, **10**, 1895–1923.

[75] Kupiec, J., Pedersen, J. and Chen, F. (1995) A Trainable Document Summarizer. *Proceedings of SIGIR* 95, Seattle, Washington, July 9–13, pp. 68–73. ACM, New York.

[76] Zhao, Z. and Liu, H. (2007) Spectral Feature Selection for Supervised and Unsupervised Learning. *Proceedings of ICML* 07, Corvallis, Oregon, USA, June 20–24, pp. 1151–1157. ACM, New York.

[77] Wu, H.C., Luk, R. W.P., Wong, K.F. and Kwok, K.L. (2008) Interpreting tf-idf term weights as making relevance decisions. *ACM Transactions on Information Systems*, **26**, 13:1–13:37.

[78] Haveliwala, T. (2003) Topic-sensitive pagerank: a context-sensitive ranking algorithm for web search. *IEEE Transactions on Knowledge and Data Engineering*, **15**, 784–796.

[79] Bettenburg, N., Premraj, R., Zimmermann, T. and Kim, S. (2008) Duplicate Bug Reports Considered Harmful . . . really?. *Proceedings of ICSM* 08, Beijing, China, September 28–October 4, pp. 337–345. IEEE Computer Society, Los Alamitos, CA.

[80] Salton, G., Wong, A. and Yang, C.S. (1975) A vector space model for automatic indexing. *Communications of the ACM*, **18**, 613–620.

[81] Tata, S. and Patel, J.M. (2007) Estimating the selectivity of tf-idf based cosine similarity predicates. *SIGMOD Record*, **36**, 7–12.

[82] Wei, X. and Croft, W.B. (2006) Lda-Based Document Models for Ad-hoc Retrieval. *Proceedings of SIGIR* 06, Seattle, Washington, August 6–11, pp. 178–185. ACM, New York.

[83] Ruggieri, S. (2002) Efficient c4.5 [classification algorithm]. *IEEE Transactions on Knowledge and Data Engineering*, **14**, 438–444.

[84] Krishnapuram, B., Carin, L., Figueiredo, M. and Hartemink, A. (2005) Sparse multinomial logistic regression: fast algorithms and generalization bounds. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, **27**, 957–968.

[85] Xia, X., Lo, D., Wen, M., Shihab, E. and Zhou, B. (2014) An Empirical Study of Bug Report Field Reassignment. *Proceedings of CSMR-WCRE* 14, Antwerp, Belgium, February 3–6, pp. 174–183. IEEE Computer Society, Los Alamitos, CA.

[86] Magerman, D.M. (1995) Statistical Decision-Tree Models for Parsing. *Proceedings of ACL* 95, Cambridge, Massachusetts, USA, June 26–30, pp. 276–283. Association for Computational Linguistics, Stroudsburg, PA.

[87] Ham, J., Chen, Y., Crawford, M. and Ghosh, J. (2005) Investigation of the random forest framework for classification of hyperspectral data. *IEEE Transactions on Geoscience and Remote Sensing*, **43**, 492–501.

[88] Kim, S.-B., Han, K.-S., Rim, H.-C. and Myaeng, S.H. (2006) Some effective techniques for Naive Bayes text classification. *IEEE Transactions on Knowledge and Data Engineering*, **18**, 1457–1466.

[89] Schneider, K.-M. (2003) A Comparison of Event Models for Naive Bayes Anti-spam e-mail Filtering. *Proceedings of EACL* 03, Budapest, Hungary, April 12–17, pp. 307–314. Association for Computational Linguistics, Stroudsburg, PA.

[90] Zhang, M.-L. and Zhou, Z.-H. (2007) Ml-knn: a lazy learning approach to multi-label learning. *Pattern Recognition*, **40**, 2038–2048.

[91] Cherkassky, V. and Ma, Y. (2004) Practical selection of SVM parameters and noise estimation for SVM regression. *Neural Networks*, **17**, 113–126.

[92] Specht, D.F. (1991) A general regression neural network. *IEEE Transactions on Neural Networks*, **2**, 568–576.

[93] Sharma, A., Koh, C., Imoto, S. and Miyano, S. (2011) Strategy of finding optimal number of features on gene expression data. *Electronics Letters*, **47**, 480–482.

[94] Zhou, J. and Zhang, H. (2012) Learning to Rank Duplicate Bug Reports. *Proceedings of CIKM* 12, Maui, HI, October 29–November 02, pp. 852–861. ACM, New York.

[95] Zimmermann, T., Nagappan, N., Guo, P.J. and Murphy, B. (2012) Characterizing and Predicting which Bugs Get Reopened. *Proceedings of ICSE* 12, Zurich, Switzerland, June 2–9, pp. 1074–1083. IEEE, Piscataway, NJ.

[96] Hofmann, T. (1999) Probabilistic Latent Semantic Indexing. *Proceedings of SIGIR* 99, Berkeley, CA, August 15–19, pp. 50–57. ACM, New York.

[97] Friedman, N., Nachman, I. and Peér, D. (1999) Learning Bayesian Network Structure from Massive Datasets: The 'Sparse Candidate' Algorithm. *Proceedings of UAI* 99, Stockholm, Sweden, July 30–August 1, pp. 206–215. Morgan Kaufmann, Burlington, MA.

[98] Kempe, D., Kleinberg, J. and Tardos, E. (2003) Maximizing the Spread of Influence through a Social Network. *Proceedings of KDD* 03, Washington, August 24–27, pp. 137–146. ACM, New York.

[99] Zhai, C. and Lafferty, J. (2001) Model-Based Feedback in the Language Modeling Approach to Information Retrieval. *Proceedings of CIKM* 01, Atlanta, Georgia, November 5–10, pp. 403–410. ACM, New York.

[100] Deerwester, S.C., Dumais, S.T., Landauer, T.K., Furnas, G.W. and Harshman, R.A. (1990) Indexing by latent semantic analysis. *JASIS*, **41**, 391–407.

[101] Liu, X. and Croft, W.B. (2004) Cluster-Based Retrieval Using Language Models. *Proceedings of SIGIR* 04, Sheffield, UK, July 25–29, pp. 186–193. ACM, New York.

[102] D'Ambros, M., Lanza, M. and Robbes, R. (2012) Evaluating defect prediction approaches: a benchmark and an extensive comparison. *Empirical Software Engineering*, **17**, 531–577.

[103] Robertson, S.E. and Jones, K.S. (1994) *Simple, Proven Approaches to Text Retrieval*. University of Cambridge, UK.

[104] Ngiam, J., Khosla, A., Kim, M., Nam, J., Lee, H. and Ng, A.Y. (2011) Multimodal Deep Learning. *Proceedings of ICML 11*, Bellevue, Washington, June 28–July 2, pp. 689–696. Omnipress, Madison, WI.

[105] Shin, H.-C., Lu, L., Kim, L., Seff, A., Yao, J. and Summers, R.M. (2015) Interleaved Text/Image Deep Mining on a Very Large-Scale Radiology Database. *Proceedings of CVPR* 15, Boston, MA, June 7–12, pp. 1090–1099. IEEE, Piscataway, NJ.

[106] Bengio, Y., Courville, A. and Vincent, P. (2013) Representation learning: a review and new perspectives. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, **35**, 1798–1828.

[107] Hooimeijer, P. and Weimer, W. (2007) Modeling Bug Report Quality. *Proceedings of ASE* 07, Atlanta, Georgia, November 5–9, pp. 34–43. ACM, New York.

[108] Bettenburg, N., Just, S., Schröter, A., Weiss, C., Premraj, R. and Zimmermann, T. (2008) What Makes a Good Bug Report?. *Proceedings of FSE* 08, Atlanta, Georgia, November 9–14, pp. 308–318. ACM, New York.

[109] Perry, D.E., Staudenmayer, N. and Votta, L.G. *et al.* (1994) People, organizations, and process improvement. *IEEE Software*, **11**, 36–45.

[110] Ko, A.J., DeLine, R. and Venolia, G. (2007) Information Needs in Collocated Software Development Teams. *Proceedings of ICSE* 07, Minneapolis, MN, May 20–26, pp. 344–353. IEEE Computer Society, Los Alamitos, CA.

[111] Bhattacharya, P., Ulanova, L., Neamtiu, I. and Koduru, S.C. (2013) An Empirical Analysis of Bug Reports and Bug Fixing in Open Source Android Apps. *Proceedings of CSMR* 13, Genova, Italy, March 5–8, pp. 133–143. IEEE Computer Society, Los Alamitos, CA.

[112] Zhou, B., Neamtiu, I. and Gupta, R. (2015) A Cross-Platform Analysis of Bugs and Bug-fixing in Open Source Projects: Desktop vs. Android vs. ios. *Proceedings of EASE* 15, Nanjing, China, April 27–29, pp. 7:1–7:10. ACM, New York.

[113] Binkley, D. (2007) Source Code Analysis: A Road Map. *Proceeduings of FOSE* 07, Minneapolis, Minnesota, May 23–25, pp. 104–119. IEEE Computer Society, Los Alamitos, CA.

[114] Zheng, W., Zou, L., Lian, X., Yu, J.X., Song, S. and Zhao, D. (2015) How to Build Templates for rdf Question/Answering: An Uncertain Graph Similarity Join Approach. *Proceedings of SIGMOD* 15, Melbourne, Victoria, Australia, 31 May–June 4, pp. 1809–1824. ACM, New York.

[115] Johansson, F.D. and Dubhashi, D. (2015) Learning with Similarity Functions on Graphs using Matchings of Geometric Embeddings. *Proceedings of KDD* 15, Sydney, NSW, Australia, August 10–13, pp. 467–476. ACM, New York.