applied sciences

MDPI

*Review*

# A Survey on Bug Deduplication and Triage Methods from Multiple Points of View

**Cheng Qian *, Ming Zhang, Yuanping Nie, Shuaibing Lu and Huayang Cao ***

National Key Laboratory of Science and Technology on Information System Security, Beijing 100085, China; zm13@alumni.sjtu.edu.cn (M.Z.); yuanpingnie@nudt.edu.cn (Y.N.); lushuaibing@alumni.sjtu.edu.cn (S.L.)
* Correspondence: qiancheng@nudt.edu.cn (C.Q.); hycao@nudt.edu.cn (H.C.)

**Abstract:** To address the issue of insufficient testing caused by the continuous reduction of software development cycles, many organizations maintain bug repositories and bug tracking systems to ensure real-time updates of bugs. However, each day, a large number of bugs is discovered and sent to the repository, which imposes a heavy workload on bug fixers. Therefore, effective bug deduplication and triage are of great significance in software development. This paper provides a comprehensive investigation and survey of the recent developments in bug deduplication and triage. The study begins by outlining the roadmap of the existing literature, including the research trends, mathematical models, methods, and commonly used datasets in recent years. Subsequently, the paper summarizes the general process of the methods from two perspectives—runtime information-based and bug report-based perspectives—and provides a detailed overview of the methodologies employed in relevant works. Finally, this paper presents a detailed comparison of the experimental results of various works in terms of usage methods, datasets, accuracy, recall rate, and F1 score. Drawing on key findings, such as the need to improve the accuracy of runtime information collection and refine the description information in bug reports, we propose several potential future research directions in the field, such as stack trace enrichment and the combination of new NLP models.

**Keywords:** bug deduplication; bug triage; information retrieval; machine learning; bug report; software development



**Citation:** Qian, C.; Zhang, M.; Nie, Y.; Lu, S.; Cao, H. A Survey on Bug Deduplication and Triage Methods from Multiple Points of View. *Appl. Sci.* **2023**, *13*, 8788. https://doi.org/10.3390/app13158788

Academic Editor: Adamu I. Abubakar

Received: 6 June 2023
Revised: 23 July 2023
Accepted: 28 July 2023
Published: 29 July 2023



## 1. Introduction

The development and operation of software is always accompanied by bugs. For example, the Mozilla project generates at least 300 bugs every day [1]. Current software development cycles are getting shorter and shorter, meaning the software is not adequately tested. In order to improve the robustness and usability of products, some institutions maintain a large-scale bug repository and bug tracking system (BTS); then, users or testers can input some features and parameters of bugs into the system so that it is easier and faster to track and fix bugs. Some open-source projects invest a lot of manpower and energy to fix bugs, e.g., Mozilla, Microsoft, Chromium, etc. There are also some open-source bug tracking systems, such as Bugzilla, GitHub issues, and JIRA, that focus on bug process management, as shown in Figure 1. Previous studies have conducted attribute analyses on different open-source bug repositories and identified unique characteristics in terms of the number of bugs, contributing developers, and other factors [2]. Additionally, some works have highlighted the high cost associated with analyzing bug reports in large-scale repositories. For example, a researcher may require a significant investment of USD 40,000 and two weeks of uninterrupted work to perform a comprehensive analysis of similarity across four bug repositories [3]. This underscores the importance and necessity of bug triage (classification of bugs according to some attributes) and deduplication (removal of the same or similar bugs) in order to address the challenges posed by large-scale bug repositories. Some organizations even encourage users to find and submit bugs in a paid

scheme. For example, Uber provides a reward of USD 600 for each user who finds a bug. In the field of hardware testing, bug triage is also very helpful [4,5] and even more necessary because the attachment of hardware bugs may be affected by more uncontrollable environmental factors, which makes the bugs quite difficult to reproduce, and the extracted features are totally different from those of the software.

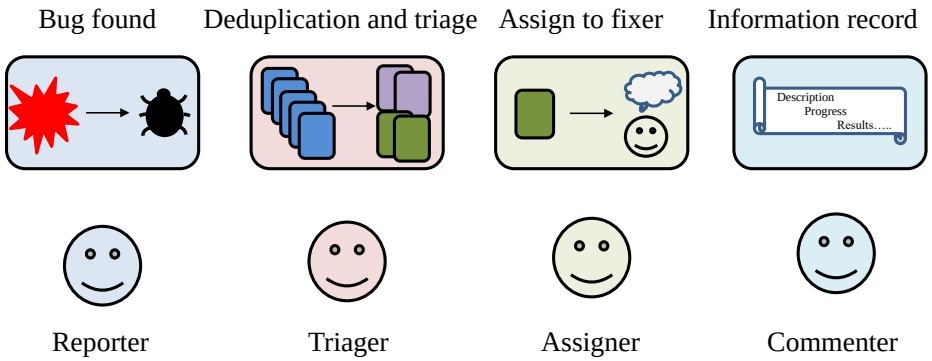


**Figure 1.** Bug management process.

Generally, project developers run vulnerability discovery tools, such as fuzzing tools, for a long time to explore the vulnerability of the system. It is an obvious fact that vulnerability fixers face a large number of bugs to deal with. However, there are many bugs that are duplicates; a survey claims that 12% of bug reports are duplicates, on average. As far as the bug reports generated by the Linux kernel are concerned, nearly ∼50% are duplicated [6]. There are many factors to consider in identifying bug deduplication, such as calling the same crash function, triggering crashes of the same type, etc. An intuitive bug deduplication and triage method is to perform bug recovery and replay. For example, methods such as symbolic execution, state space search, and taint analysis can be used to perform bug recovery and replay, but these approaches have quite high runtime monitoring costs.

Unlike deduplication, bug triage considers factors including severity, bug platform, theme, semantic/syntactic/functional similarity, etc. At present, it is difficult to ensure that bug-fixing tasks are assigned to the most suitable developers. Some work [7] mentioned that about half of the bugs in some large open-source projects are tossed, which means that expertise cannot solve the assigned bug, indicating low efficiency of bug fixing. Bug triage has another connotation in some cases: setting reducing the bug fix time cost as the primary optimization point, regardless of the similarity between bugs. For example, the goal of [8] was to distribute bugs to the right developers at the right time. Considering random errors that may occur at any time and the specific reality of developers who plan to change at any time, this work is based on the Markov decision process model (MDP), using approximate dynamic programming (ADP) to minimize entropy, with the optimization goal of minimizing bug repair time decision making. This is not aligned with the topic of our article and is therefore beyond the scope of discussion.

Bug deduplication and bug triage play a crucial role in improving the efficiency of bug fixes. The former helps reduce redundancy in bugs, alleviating the current situation of a large number of bug submissions awaiting analysis. The latter classifies bugs based on certain logical similarities and transfers them to the corresponding developers. This approach maximizes the chances of pushing bugs to potential experts who can effectively improve bug-fixing efficiency.

Bug deduplication and bug triage can be performed based on two types of information. The first type is the bug report, which includes the bug's category, the platform it occurred on, the components involved, priority, the personnel involved, and some real-time records. Typical bug reports, as shown in Figure 2, are generally semistructured files that contain various information [9,10]. The content includes the context of the crashing thread, such as the stack trace and processor registers, as well as a subset of the machine memory contents at the time of the crash. It also includes basic information about the bug, such as the system,

version, system components, description, fix comments, and a sequence of developers who may have worked on the bug (tossing sequence), among other details [11].

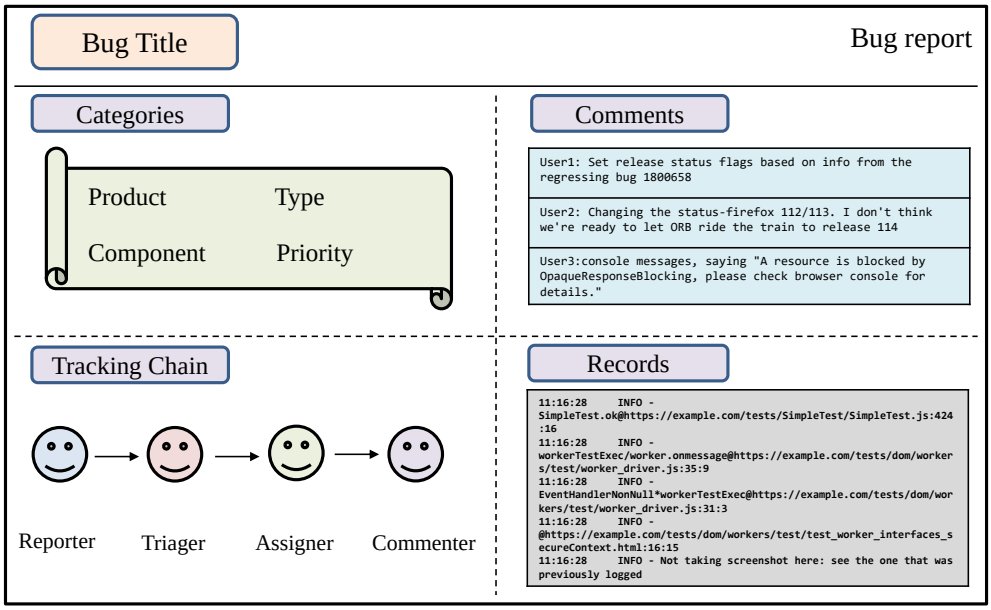


**Figure 2.** Information contained in a typical bug report.

The life cycle of a bug report is illustrated in Figure 3. Once a bug is generated, its corresponding bug report is generated, deduplicated, and triaged. It is initially marked as "unconfirmed", then assigned to an appropriate developer at the right time (referred to as triage). If the developer cannot fix the bug correctly, the bug report is tossed back into the bug pool and remarked as "unconfirmed". Otherwise, it is marked as "resolved" and awaits verification. Once a verifier confirms that the bug has been fixed, the bug report is marked as "verified", and the case is closed. If it is not confirmed, the bug report is tossed.

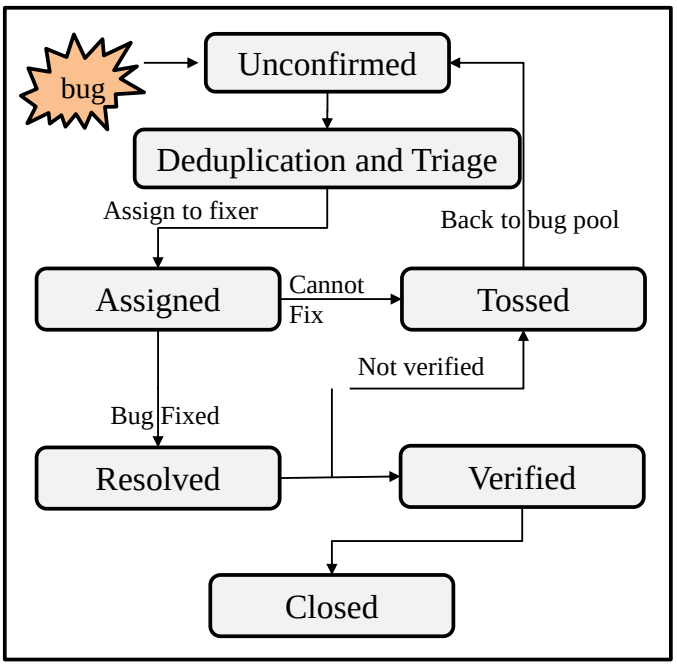


**Figure 3.** The life cycle of a bug report.

The second type is the runtime information triggered by the bug, such as stack trace, coverage, control flow, crash point, etc. Achieving precise bug deduplication and triage is challenging. Both concepts rely on similarity, with deduplication primarily focusing on content similarity, while triage may also consider the characteristics of the fixer. Figure 4 illustrates the process of feature extraction and similarity calculation for bug reports, taking bug reports as an example. The upper part of the figure shows a process using information retrieval methods. It involves feature extraction using topic modeling techniques, followed by similarity calculation using mathematical models such as Cosine similarity and Jaccard similarity. The lower part of the figure presents a process using machine learning methods. It starts by extracting the control flow graph from the crash code snippets in the bug report. The control flow graph is then used as input to a neural network to compute the likelihood of duplicated or similar bug reports.

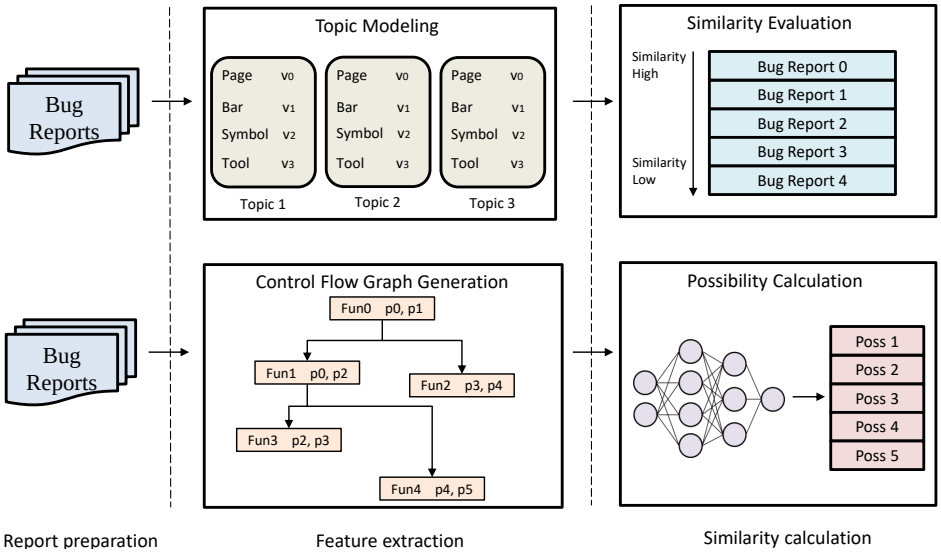


**Figure 4.** Process of feature extraction and similarity calculation for bug reports.

Bug deduplication and bug triage have been hot topics in recent years. Figure 5 illustrates the trend of related works in recent years, showing an increasing number of studies in this area. We conducted a survey of relevant works, summarized the research methods and their effectiveness, analyzed the drawbacks of current approaches, and proposed possible research directions for the future.

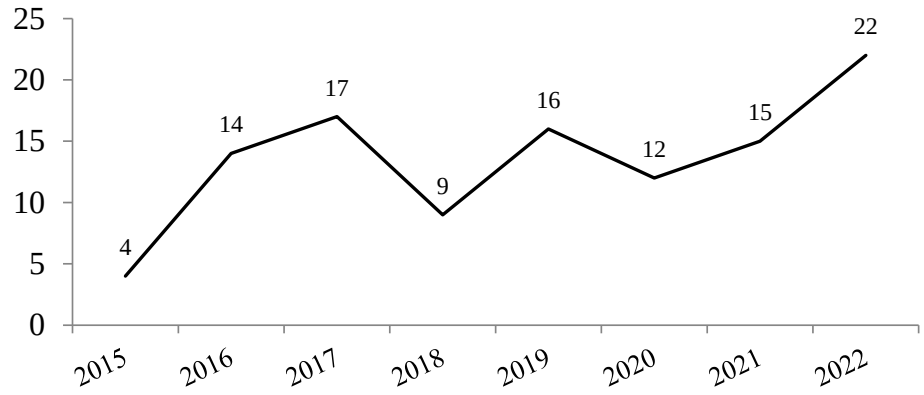


**Figure 5.** The number and trend of bug-deduplication- and triage-related work in recent years.

This paper focuses on the following research questions:

1. What is the roadmap of deduplication- and triage-related work? What mathematical methods are commonly used to address these problems?
2. What are the main approaches currently used for deduplication and triage? What are the recent works on each approach and how are they implemented?
3. What datasets are used in the related works? How are these works evaluated, and what are their actual results?
4. What conclusions can be drawn from the current works? What are the potential research directions for the future?

The main contributions of this paper are as follows:

1. This paper summarizes the mathematical concepts and methods that are commonly used by bug deduplication and triage methods.
2. This paper summarizes relevant works based on runtime information and analyzes the commonly used technical approaches from three perspectives. This paper has provides a comparison of the implementation methods and results of each work.
3. This paper summarizes relevant works based on bug reports and explains the technical principles from two perspectives: information retrieval and machine learning. This paper provides detailed descriptions of the implementation approaches of various methods and a comparative analysis of their performance differences.
4. This paper draws some empirical findings and proposes some possible future research points in terms of bug deduplication and triage.

The remainder of this paper is organized as follows. Section 1 introduces the motivation, brief content, and contribution of the article Section. 2 discusses related surveys and compares them with this paper. Section 3 describes the roadmap of existing literature and background knowledge commonly used in existing works. Section 4 summarizes the related research based on runtime information. Section 5 reviews the related works on bug reports. Section 6 lists the datasets and evaluation methods used in the work and analyzes the effectiveness of related works. Section 7 illustrates the existing issues in current methods and proposes possible research directions for the future. Section 8 provides a conclusion for the entire paper.

## 2. Related Survey

Many existing research works on bug deduplication and triage primarily focus on bug reports. These methods generally require the involvement of domain experts, and automated methods have shown limited accuracy.

Neysiani et al. compared IR-based and ML-based methods for bug report deduplication [12], and the experimental results showed no significant difference in terms of accuracy or runtime efficiency. Campbell et al. conducted a quantitative analysis of commonly used bug classification methods, including signature-based approaches (such as functions, addresses, and linked libraries) and text-tokenized methods. The results indicated that IR methods based on TF-IDF had better triage effectiveness. Udden et al. surveyed bug prioritization works based on bug reports up to 2017 [13], covering various methods, such as data mining and machine learning techniques for bug identification, clustering, classification, and rating, including supervised and unsupervised methods. The advantages and limitations of the methods were analyzed, with the authors concluding that there is still room for improvement in current approaches. However, this work is relatively early, focusing only on bug prioritization using bug reports. Our work primarily focuses on research conducted after 2015 to ensure the survey's relevance and up-to-dateness. Sawant et al. conducted a survey on bug classification work based on bug reports [14], summarizing various related techniques, such as text-based classification, recommendation-based approaches, and tossing-graph-based methods. Neysiani et al. summarized commonly used features and general steps in bug report deduplication [15], highlighting the existing issues and potential research points for optimization. However, the coverage of these articles is not comprehensive, and they do not specifically focus on runtime information for classifica-

tion. Yadav et al. surveyed classification methods based on machine learning, profiles, or metadata, comparing and discussing the pros and cons of different approaches [16]. They concluded that no single method has advantages in all dimensions and provided insights into potential research points such as the cold activation problem and load balancing.

Chhabra et al. briefly described the factors to consider in bug triage and listed methods and contributions of some bug triage-related works [17]. Neysiani et al. provided a general description of the processes for IR-based and machine-learning-based methods [18], along with relevant works listed separately. Lee et al. surveyed deduplication methods based on natural language processing (NLP) [7], information retrieval (IR), and clustering, as well as classification techniques using naive Bayes combined with machine learning. They suggested focusing on improving the efficiency of deep learning models for precise bug report classification in future research. These works primarily focus on investigating specific categories of techniques, while our work encompasses both common information retrieval (IR) and machine learning (ML)-based techniques, providing a comprehensive investigation into both categories of approaches.

Pandey et al. quantitatively analyzed bug triage using common machine learning methods and tested six approaches [19], including SVM, naive Bayes, and random forest. The experimental results showed that SVM performed better in terms of F-measure, average accuracy, and weighted average F-measure metrics. However, it is worth noting that this article focused on a binary classification problem, which differs from bug triage.

Goyal et al. mainly compared IR and machine learning methods for bug triage [20]. They summarized over seventy related articles to discuss the pros and cons of IR and machine learning methods. Through experiments on multiple open-source projects, they found that IR-based bug triage methods had better performance. However, compared to our work, this study is relatively old and does not include recent developments. It also focuses solely on bug triage based on bug reports, lacking research on runtime information and deduplication. The previous works mainly investigated relevant research related to bug reports as the target. In contrast, our work includes not only bug reports but also research related to runtime information as the subject of analysis, making it more comprehensive. A comparison between the related surveys and our work is presented in Table 1.

**Table 1.** Comparison between related survey and our work.

| Cited Paper | Study on Commonly Used Methods | Study on Runtime Information-Based Approaches | Study on Information Retrieval Approaches | Study on Machine Learning Approaches |
|---|---|---|---|---|
| Neysiani et al. [12] | Not included | Not included | Bug report deduplication using IR methods | Bug report deduplication using ML methods |
| Udden et al. [13] | Not included | Not included | Bug prioritization using data mining | Bug prioritization using machine learning |
| Sawant et al. [14] | Not included | Not included | Bug report classification using text-based analysis, recommendation, etc. | Not included |
| Neysiani et al. [15] | Not included | Not included | Not included | Features and general steps for bug report deduplication |
| Yadav et al. [16] | Not included | Not included | Not included | Comparison of ML-based classification |
| Chhabra et al. [17] | Not included | Not included | Factors to consider in bug triage | Not included |

**Table 1.** *Cont.*

| Cited Paper | Study on Commonly Used Methods | Study on Runtime Information-Based Approaches | Study on Information Retrieval Approaches | Study on Machine Learning Approaches |
|---|---|---|---|---|
| Neysiani et al. [18] | Not included | Not included | General description of the IR-based methods | General description of the ML-based methods |
| Lee et al. [7] | Not included | Not included | Deduplication using IR methods | Deduplication using NLP, naive Bayes, etc. |
| Pandey et al. [19] | Not included | Not included | Not included | Bug triage using six ML models |
| Goyal et al. [20] | Not included | Not included | Bug triage using IR methods | Bug triage using ML methods |
| Our work | Feature extraction methods and similarity calculation methods | Deduplication and triage based on runtime stacks, coverage, and context | Deduplication and triage based on texture analysis, topic modeling, etc. | Deduplication and triage based on CNN, LSTM, transformer, etc. |

## 3. The Roadmap of Existing Literature

### *3.1. Overview of Relevant Literature in Recent Years*

Deduplication and triage fundamentally involve considering the similarity between bugs; therefore, the extraction of bug features and the calculation of similarity are the main research topics. As shown in Figure 6, the evolution of techniques used in related works exhibits a clear temporal pattern. Around 2010, traditional text matching and machine learning methods were the mainstream approaches for determining similarity. Traditional text matching methods primarily used dynamic programming-based techniques such as longest common subsequence and longest common substring. Meanwhile, machine learning methods at that time mainly relied on SVM, naive Bayes, and other classification models. Some representative works during this period include [21–23].

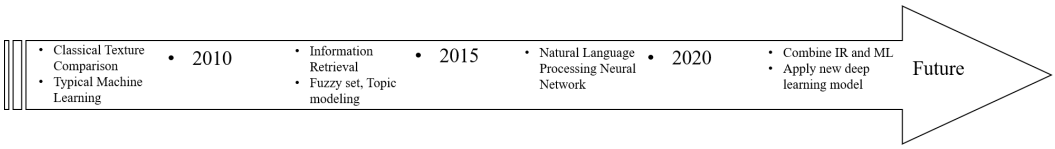


**Figure 6.** How the state-of-the-art deduplication and triage techniques evolved.

Around 2015, information retrieval methods started to be extensively developed and used, including topic modeling, fuzzing set, and text feature extraction. These methods can more accurately model bug reports, extract feature vectors, and measure the distance between texts using similarity calculation techniques. Some typical works during this period include [24–26].

After 2015, thanks to the rapid advancement of deep learning methods, especially various neural network models, such as NLP-based models, adjusted deep learning methods showed outstanding effects in large-scale similarity analysis. Other typical neural network models, such as CNNs, have also been applied in feature extraction. Some representative works during this period include [27–29].

The future development of related techniques may follow two directions. First, combining the strengths of IR and ML for deduplication and triage may be a promising approach. IR excels in accurate feature extraction, while ML outperforms in similarity analysis, recommendation, and prediction. Some works have already proposed methods based on this idea [30,31]. Secondly, optimizing the application of the latest advances in machine learning and information retrieval, such as transformer models used in large-scale language processing, after appropriate transfer learning may lead to better results than the current

models. These potential research directions are discussed in detail in Sections 4 and 5 of this paper.

Figure 7 illustrates some of the recent works in the field of deduplication and triage, including approaches based on runtime information, information retrieval methods for bug reports, and machine learning approaches for bug reports. Overall, deduplication and triage have been hot research topics in recent years. In terms of research quantity, the majority of works have focused on deduplication and triage based on bug reports. This is mainly because bug reports are commonly used as the medium for storing bugs in bug tracking systems (BTS), making them more readily available. On the other hand, works utilizing runtime information face higher difficulty and complexity, as they require collection and analysis of various pieces of information related to crashes. Among the bug-report-based works, information-retrieval-based approaches show a relatively steady distribution over the years, while machine-learning-based approaches have seen a significant increase in recent years due to the rapid development of machine learning techniques.

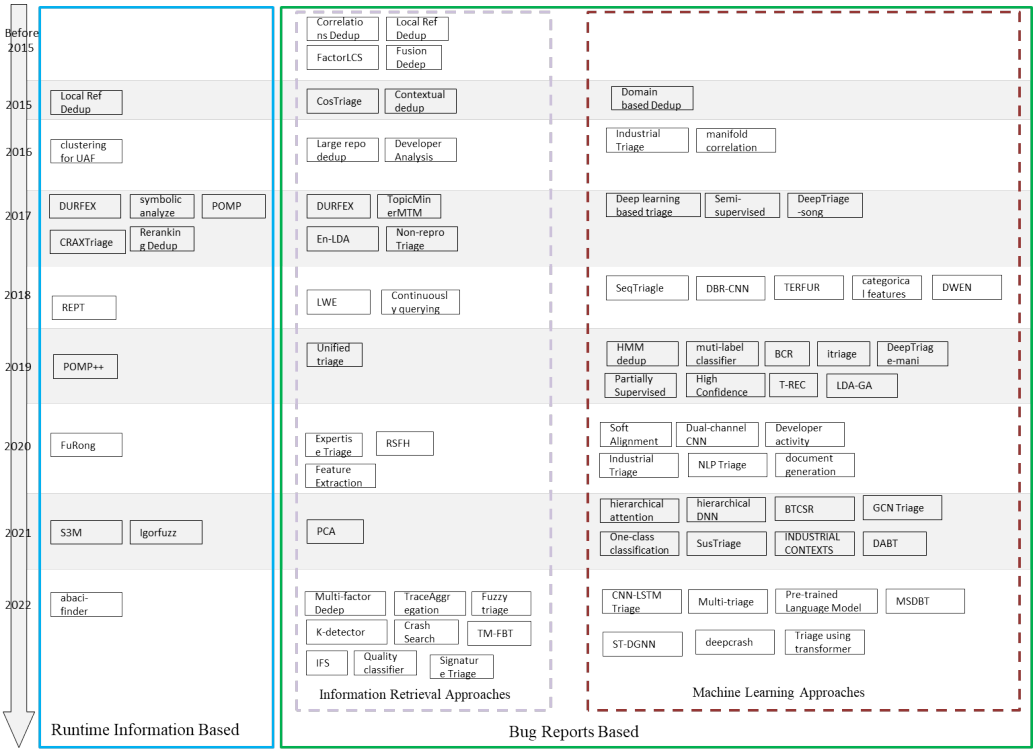


**Figure 7.** Overview of related work in recent years.

## *3.2. Background Knowledge*

### 3.2.1. Feature Extraction and Selection

Regardless of whether using information retrieval (IR) or machine learning methods, topic modeling of bug reports is a commonly used approach for feature extraction. Such methods aim to abstract a piece of text into several keywords that can represent the entire text. Common feature extraction models for natural language processing include N-Gram, LDA, TF-IDF, and others.

(1) TF-IDF

The term frequency inverse document frequency (TF-IDF) method is used to extract textual features from bug reports. It quantifies the importance of a term within a document by calculating the frequency of the term in the document ($TF(t, D)$) and the number of documents that contain the term ($IDF(t)$).

Let $DF(t)$ represent the terminology and t represent the number of documents in which the terminology appears at least once. Let $|D|$ represent the total number of reports. Formula (1) represents the calculation process for *IDF* (inverse document frequency):

$$IDF(t) = \log \frac{|D|}{DF(f)} \tag{1}$$

The eigenvalue ($w_i$) belonging to $t$ can be expressed as:

$$TF\_IDF(t) = w_i = TF(t, D) * IDF(t) \tag{2}$$

Finally, the $TF\_IDF$ feature vector about terminology can be represented as:

$$\vec{V} = (W_1, W_2, W_3, \ldots, W_n) \tag{3}$$

The TF-IDF feature vectors of two different documents can be used to estimate the similarity between the texts based on cosine similarity, as Formula (4) shows.

$$cosine\_similarity = \frac{(AB)}{||A|| * ||B||} \tag{4}$$

where $A$ and $B$ represent the TF-IDF feature vectors of the two documents, $\cdot$ denotes the dot product, and $||A||$ and $||B||$ represent the Euclidean norms of the respective vectors.

(2) N-Gram

The N-Gram model is based on an $N-1$ order Markov model, which assumes that the current element (word) is only dependent on the previous $N-1$ elements. In practical applications, N is typically limited to 3 or 4 because larger values of N lead to a significant increase in complexity while providing limited performance improvement.

The N-Gram model primarily aims to predict the probability of the current element, given that the previous $N-1$ elements are known, or to evaluate the likelihood of a sentence composed of N elements. Formula (5) represents the calculation process for probability:

$$p(W_1, W_2, W_3, \ldots, W_n) = p(W_1) \cdot p(W_2|W_1) \cdot \cdots \cdot p(W_n|W_1, W_2, \ldots, W_{n-1}) \tag{5}$$

where $p(W_n|W_1, W_2, \ldots, W_{n-1})$ represents the probability of the current word ($W_n$), given the previous $N-1$ words ($(W_1, W_2, \ldots, W_{n-1})$. This probability is estimated based on the frequencies of N-grams observed in a training corpus.

(3) LDA

LDA (latent Dirichlet allocation) is an unsupervised probabilistic topic modeling technology. Essentially, it aims to infer the topic distribution of a document based on its content. To achieve this, LDA requires users to provide the expected number of topics, denoted as K, for the given bug reports. LDA utilizes the Dirichlet distribution, as shown in Equation (6), as a prior distribution. It then updates the prior distribution based on the specific distribution observed in the bug reports to form the posterior probability distribution. Gibbs sampling is then employed to infer the specific probability distribution parameters of the LDA topic model.

$$Dir(\vec{\rho}|\vec{\alpha}) = \frac{\Gamma(\sum_K^{k=1} a_k)}{\prod_K^{k=1} \Gamma(a_k)} \prod_K^{k=1} \rho_k^{a_k - 1} \tag{6}$$

where $\Gamma(n) = (n-1)!$

(4) Chi-square ($\chi^2$) test

The chi-Square test can be used to assess the correlation between a particular feature and the outcome, thereby identifying redundant or irrelevant features. It calculates the deviation between the observed and expected values for a given feature. The test determines

whether the feature is independent of the outcome based on the chi-square distribution with K degrees of freedom, as shown in Equation (7).

$$\chi^2 = \sum_{k}^{i=1} \frac{(A_i - B_i)^2}{E_i} = \sum_{k}^{i=1} \frac{(A_i - np_i)^2}{np_i} \tag{7}$$

(5) Mutual Information

Mutual information measures the association between random variables, specifically the reduction in uncertainty of variable $Y$ given variable $X$. The concept of uncertainty can be quantified using entropy. The mutual information between $X$ and $Y$ can be calculated using Equation (8), where $p(x, y)$ represents the joint probability of $x$ and $y$ occurring simultaneously.

$$I(X, Y) = \sum_{x,y} p(x, y) \log \frac{p(x, y)}{p(x)p(y)} \tag{8}$$

When $x$ represents features and $y$ represents labels or outcomes, a higher mutual information value indicates a stronger correlation or dependence between the two. It signifies that feature $x$ provides more information about label $y$; therefore, it is more relevant for predicting or explaining the target variable.

3.2.2. Similarity Evaluation Model

(1) BM25F

BM25F (Best Matching 25 with Fields) is a classical method used to measure the similarity between texts. This method calculates a score based on the frequency of occurrence of words in different weighted segments of an article. Formula (9) represents the calculation process:

$$Score(Q, d) = \sum_{k=1}^{n} \log \frac{N - n(q_i) + 0.5}{n(q_i) + 0.5} \cdot \frac{f_i^u}{k_1 + f_i^u} \tag{9}$$

where $f_i^u = \sum_u^{k=1} w_k \cdot \frac{f_{ui}}{1 - b_u + b_u \cdot \frac{ul_u}{uvul_u}}$, $f$ is a word frequency statistical function, $n$ is the number of texts containing a certain word, and $ul_u$ and $uvul_u$ represent the text segment fields.

(2) Support Vector Machine (SVM)

SVM is a supervised method that focuses on extracting feature vectors from bug reports. It can label these feature vectors and train an SVM to construct a hyperplane that maximizes the distance between feature vectors labeled with different tags. For linear SVM, the general form of its hyperplane is

$$g(x) = \omega \cdot x + b = 0 \tag{10}$$

where $x$ is an n-dimensional vector. Solving the most optimistic hyperplane is essentially solving the problem indicated by Equation (11).

$$min(\frac{1}{2}(\omega \cdot \omega) + C \sum_{i=1}^{l} \varepsilon_i) \tag{11}$$

where $y_i[(\omega \cdot x) + b] + \varepsilon_i \geq 1$ for $i$ in $1 \ldots l$, and $\varepsilon_i$ represents the slack variables in the case of linear inseparability.

Nonlinear SVM only requires replacement of the direct inner product in linear SVM with a kernel function. Widely used kernel functions include the polynomial kernel, sigmoid kernel, and radial basis function kernel.

$$Polynomial\ kernel : (\gamma x_i^T x_j + r)^d, \gamma > 0 \tag{12}$$

$$Sigmoid\ kernel : exp(-\gamma||x_i - x_j||^2), \gamma > 0 \tag{13}$$

$$rbf\ kernel : \tan h(\gamma x_i^T x_j + r), \gamma > 0 \tag{14}$$

(3) Logistic Regression

After extracting feature vectors from bug reports, one can use logistic regression to make predictions by training a model, as shown in Equation (15). The training process involves solving for the parameters in the equation to maximize the likelihood estimation for the data in the training set.

$$\begin{aligned} P(x, y, B) &= \frac{exp(\beta_k^T x)}{\sum_K^{j=1} B_j x} \\ I(B, D) &= \sum_i [\sum_k y_{ik} \beta_k^T x_i - \ln \sum_k exp(\beta_k^T x_i)] \end{aligned} \tag{15}$$

(4) Naïve Bayes

Consistent with the idea of Logistic Regression, the naïve Bayes model is constructed using the eigenvectors of bug reports as shown in Formula (16).

$$P(y|x_1, x_2, \ldots, x_d) = C \cdot P(y) \prod_{i=1}^{d} P(x_i|y) \tag{16}$$

The final calculation process is the process of determining the corresponding result of the maximum probability ($P$), expressed as

$$y = argmax_y P(y) \prod_{i=1}^{d} P(x_i|y) \tag{17}$$

Specific to the bug report containing the text, the probability of using the parameter to divide the document can be expressed as:

$$P(r_j, d_j, \theta) = \prod_{k=1}^{r} P(w_k, d_j, \theta)^{N_k} \tag{18}$$

Here, the variable $P(w_k, d_j, \theta)$ can be calculated after considering the statistical count of words and the frequency of occurrence of all words in a particular bug report. In this context, it is generally assumed that the words are independent. Otherwise, when calculating $P(w_k, d_j, \theta)$, parameters ($\lambda$) are introduced to recompute the frequency of word occurrences in the report.

(5) K-Nearest Neighbors

After obtaining the feature vector of a bug report, the KNN model calculates the distances between features using the Minkowski distance function (as shown in Equation (19)). It selects the k nearest points based on distance and assigns the majority class among those points to the current bug report.

$$dis(x, y) = \sqrt[p]{(x_0 - y_0)^p + (x_1 - y_1)^p + \cdots + (x_k - y_k)^p} \tag{19}$$

(6) Random Forest—Extreme Tree

Random Forest combines N CART decision trees and is trained using the bagging technique. Each decision tree in RF is trained independently by randomly selecting N samples with replacements. During the training process, RF randomly selects m sample features and, according to a preset strategy, chooses one of them as the splitting attribute for a node. When making predictions for a given instance, the majority classification result from the decision trees is selected.

Extreme tree, on the other hand, is an even more aggressive strategy within random forest. It uses the entire training set during the bootstrap process and randomly selects splitting attributes when constructing decision trees.

(7) Hidden Markov Model (HMM)

The hidden Markov model (HMM) can make predictions about possible states based on current observations and adjust the model's parameters using the observed and predicted results. An HMM can be described using a five-tuple,

$$HMM = (X, O, \pi, A, B) \tag{20}$$

where $X$ represents the hidden state vector, $O$ represents the observed state vector, $A$ is the hidden state transition matrix, $B$ is the confusion matrix representing the probabilities of observed states given hidden states, and $\pi$ is the initial probability vector for hidden states.

In the context of text classification, the training texts are first subjected to feature extraction to obtain features such as frequency and probability distributions. These features are used to form the initial state vector. By training the HMM with the corresponding class labels of the texts, the parameters of the HMM are determined. This trained HMM can then provide a probability distribution for classification of test texts.

(8) LSTM/CNN

Convolutional Neural networks (CNNs) are widely used in image processing and have achieved outstanding results. Recently, there have been efforts to apply CNNs to text classification tasks. The approach and model construction are similar to those used in image processing. As shown in Figure 8, the text is first converted into vectors using techniques like word2vec. Then, convolutional operations, pooling, and other operations are performed to generate probability outputs corresponding to the classification results.

LSTM (long short-term memory) is a variant of a recurrent neural network (RNN) that excels at capturing spatial dependencies in text, making it widely used in text-related fields such as natural language processing (NLP). As shown in Figure 9, LSTM is an extended version of RNN. It introduces a forget gate, which determines what information should be retained and what should be forgotten. This helps alleviate the vanishing gradient problem caused by forgetting hidden state variables in RNNs. Consequently, LSTM can achieve higher accuracy in text classification tasks. In some works, CNN and LSTM are also combined to leverage the strengths of both architectures.

Performing transfer learning based on existing models is an effective method for text classification, as it reduces the time and complexity of model construction. As shown in Figure 10, transfer learning involves fine tuning of the parameters of a pretrained model to adapt to a new optimization objective.

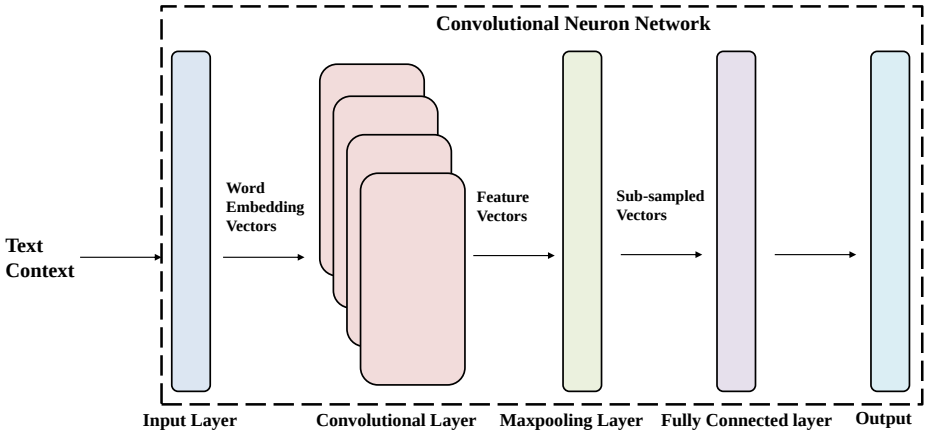


**Figure 8.** Example image of a CNN for text classification.

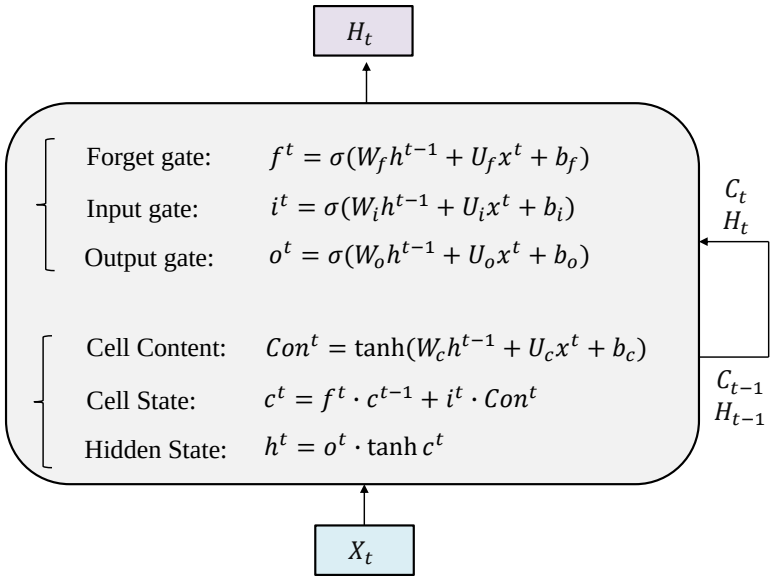


**Figure 9.** Calculation process in an LSTM unit for text classification.

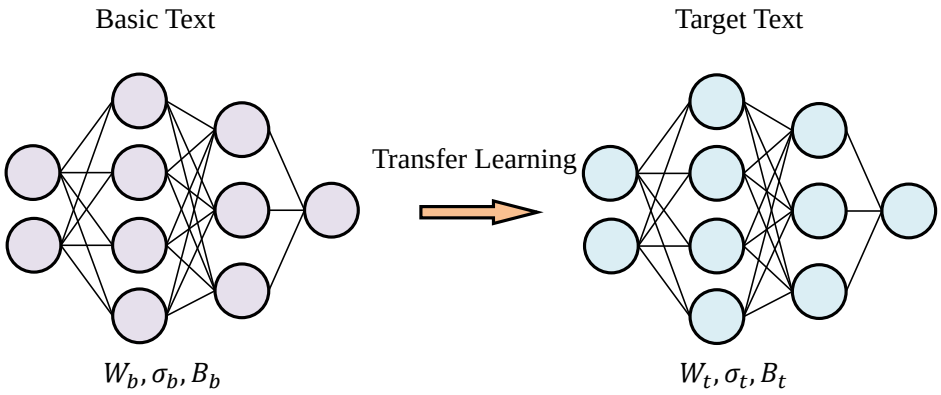


**Figure 10.** Transfer learning used to build ML models for text classification.

### *3.3. Commonly Used Datasets*

Many datasets have been used in these works, as listed in Table 2. Among them, Mozilla and Eclipse are two commonly used publicly available datasets. NetBeans, OpenOffice, and others are also popular public datasets. Around 60% of the works also utilize other public datasets, such as Jira, Apache, GCC, etc. Furthermore, some works make use of private datasets from companies for their research purposes.

**Table 2.** Datasets used in the literature.

| Dataset | Proportion in Works |
|---|---|
| Mozilla | ∼64.4% |
| Eclipse | ∼37% |
| Netbeans | ∼19.2% |
| Openoffce | ∼11% |
| Others | ∼60% |

### *3.4. Evaluation Parameters*

Most of the existing works in bug triage utilize some or all of the parameters, including accuracy, precision, recall rate, and F1 score, to measure the performance of their methods. These parameters are based on four fundamental statistical measures: true positive (TP),

true negative (TN), false negative (FN), and false positive (FP). TP refers to the cases where the actual and predicted results are consistent and both are duplicates (for deduplication) or belong to the same class (for triage). TN refers to the cases where the actual and predicted results are consistent and neither is duplicated (for deduplication) or belongs to a different class (for triage). FN refers to the cases where the actual results are duplicates or belong to the same class but the predictions are not duplicates or belong to a different class. FP refers to the cases where the actual results are not duplicates or belong to a different class but the predictions are duplicates or belong to the same class.

Based on TP, TN, FP, and FN, accuracy is defined as the measure of how well the method performs on all samples, as shown in Formula (21).

$$\text{Acc} = \frac{\text{TP} + \text{TN}}{\text{TP} + \text{TN} + \text{FN} + \text{FP}} \tag{21}$$

Precision is defined as the measure of the method's accuracy in predicting duplicates/same class, as shown in Formula (22).

$$\text{Precision} = \frac{\text{TP}}{\text{TP} + \text{FP}} \tag{22}$$

The recall rate is defined as the measure that focuses on the samples that are actually duplicates/same class, as shown in Formula (23).

$$\text{Recall Rate} = \frac{\text{TP}}{\text{TP} + \text{FN}} \tag{23}$$

F1 score combines both recall and precision and provides an overall measure of the method's performance, as shown in Formula (24).

$$\text{F-score} = \frac{2 \cdot \text{Recall} \cdot \text{Precision}}{\text{Recall} + \text{Precision}} \tag{24}$$

## 4. Works Based on Runtime Information

Runtime information, such as the core dump generated when a bug occurs, can be utilized as bug features. This includes the crash-time register state, memory management information, stack pointers, and more. With the advancements in hardware-assisted information collection methods, developers can now gather more instruction-level details using technologies like Intel PT during runtime.

### *4.1. Methods Based on Comparing Stack Trace*

One of the most classic parameters used to determine whether a bug is a duplicate is the stack trace. In general, stack trace has attributes including ID, timestamp, and a series of function call records (also known as the frame) before the error occurred. Figure 11 shows an example of frames in a stack trace from Eclipse. When a program crashes, it stores several pieces of function call information on the runtime stack before the crash, which can be extracted and used to identify the uniqueness of the bug. Some studies claim that 80% of causes can be found in the stack trace hash [32].

```
org.eclipse.equinox.p2.core.ProvisionException: No repository found at http://download.eclipse.org/webtools/downloads/drops/R3.8.0/S-3.8.0M5-20160202064558/repository/.
        at org.eclipse.equinox.internal.p2.repository.helpers.AbstractRepositoryManager.fail(AbstractRepositoryManager.java:395)
        at sun.reflect.NativeMethodAccessorImpl.invoke0(Native Method)
        at sun.reflect.NativeMethodAccessorImpl.invoke(NativeMethodAccessorImpl.java:62)
        at sun.reflect.DelegatingMethodAccessorImpl.invoke(DelegatingMethodAccessorImpl.java:43)
        at java.lang.reflect.Method.invoke(Method.java:483)
        at org.eclipse.oomph.util.ReflectUtil.invokeMethod(ReflectUtil.java:117)
        at org.eclipse.oomph.p2.internal.core.CachingRepositoryManager.fail(CachingRepositoryManager.java:353)
        at org.eclipse.oomph.p2.internal.core.CachingRepositoryManager.loadRepository(CachingRepositoryManager.java:244)
        at org.eclipse.oomph.p2.internal.core.CachingRepositoryManager$Metadata.loadRepository(CachingRepositoryManager.java:476)
        at org.eclipse.equinox.internal.p2.metadata.repository.MetadataRepositoryManager.loadRepository(MetadataRepositoryManager.java:96)
        at org.eclipse.equinox.internal.p2.metadata.repository.MetadataRepositoryManager.loadRepository(MetadataRepositoryManager.java:92)
        at org.eclipse.equinox.internal.p2.metadata.repository.CompositeMetadataRepository.addChild(CompositeMetadataRepository.java:166)
        at org.eclipse.equinox.internal.p2.metadata.repository.CompositeMetadataRepository.<init>(CompositeMetadataRepository.java:106)
        at org.eclipse.equinox.internal.p2.metadata.repository.CompositeMetadataRepositoryFactory.load(CompositeMetadataRepositoryFactory.java:122)
        at org.eclipse.equinox.internal.p2.metadata.repository.MetadataRepositoryManager.factoryLoad(MetadataRepositoryManager.java:57)
        at org.eclipse.equinox.internal.p2.repository.helpers.AbstractRepositoryManager.loadRepository(AbstractRepositoryManager.java:768)
        at sun.reflect.GeneratedMethodAccessor60.invoke(Unknown Source)
        at sun.reflect.DelegatingMethodAccessorImpl.invoke(DelegatingMethodAccessorImpl.java:43)
```

**Figure 11.** Frames in a stack trace from Eclipse.

Several approaches utilize or build upon the classic longest common subsequence algorithm to measure the similarity between stack traces and determine if they are duplicate bugs (known as "rebucketing") [33]. While this method is simple and easy to use, it suffers from low robustness. In 2022, Rodrigues et al. proposed TraceSim, which combines TF-IDF and machine learning methods for bug deduplication. TraceSim first uses the Needleman–Wunsch algorithm to determine the global optimal alignment by considering the position and frequency of each stack trace frame to assign weights. The parameters used for weight calculations are learned and adjusted using a Bayesian hyperparameter optimizer called TPE. Ultimately, TraceSim [34] calculates a similarity score between two stack traces. The alignment process is illustrated in Figure 12.

To enhance the practicality of alignment methods, some works have proposed accelerated techniques for commonly used stack trace alignment methods. One example is FaST, which achieves linear time complexity. Additionally, there are current efforts to analyze stack traces using LSTM, which can extract temporal features. These advancements aim to improve the efficiency and accuracy of bug deduplication by utilizing stack traces and exploring different alignment techniques [35]. Dunn et al. proposed the use of GPU acceleration to speed up the cosine similarity comparison, as well as calculation of longest common subsequence, longest common substring, and other operations required for bug triage [36]. The implementation results on approximately 1.8 million bug reports from Eclipse and Mozilla showed that using GPU acceleration can improve the speed of bug triage by nearly 100 times.

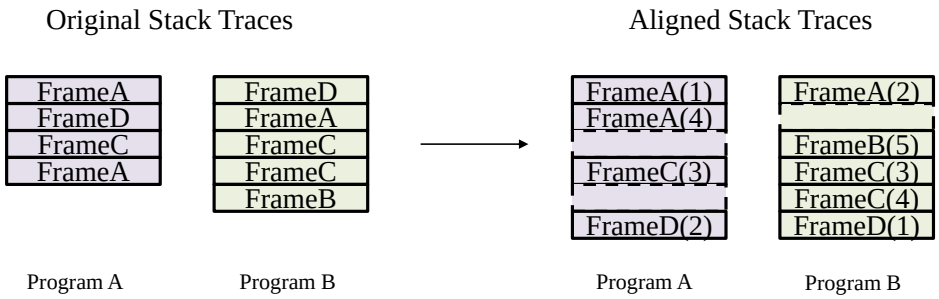


**Figure 12.** Alignment process of stack traces.

The stack trace can also be used analyze and locate potential causes of vulnerabilities, aiding in deduplication or triage processes. CrashLocator is a technique that extends the stack trace information by recording the minimum number of times each function is called throughout the execution leading up to a crash. It then expands the stack trace information and performs spectrum analysis on the relevant functions marked in the crash execution path to calculate a suspicion score for each function. The functions with the highest suspicion scores can be attributed as potential bug causes, thereby facilitating effective classification. CrashLocator served as a valuable tool in identifying and categorizing bugs based on their stack trace information [37].

Koopaei et al. proposed CrashAutomata [38], which utilizes the stack trace of a bug to extract variable-length N-gram features, overcoming scalability issues. They obtained trace patterns of lengths of 1 to n and constructed an automaton to classify bugs. Experimental results showed an average F measure of 97%. Sabor et al. introduced a multistep feature extraction method called DURFEX [39]. It reduced the number of features by replacing function names, then used variable-length N-grams to extract feature vectors. They constructed an adjacency matrix based on multiple feature vectors. For new bug reports, they extracted feature vectors and compared them individually with the feature vectors in the adjacency matrix to calculate similarity, thereby determining if there were bug duplication issues. The advantage of this method lies in its reduced execution time. Compared to using only 1-g functions, DURFEX can reduce execution time by 93%, and using 2-g can reduce it by up to 70%. Tian et al. proposed FuRong [40], a tool for deduplicating and merging Android bugs into presentable bug reports. This tool initially classifies bugs using

pretrained decision trees and naive Bayes classifiers. It then extracts features from the bug's LogCat log and measures the similarity of the stack trace using the Levenshtein distance to determine if the bugs are similar, enabling deduplication. The article evaluated bug classification and reported an average bug classification precision of 93.4% and an average classification accuracy of 87.9%.

Khvorov et al. proposed S3M [41], a machine learning model based on Siamese architecture, for calculation of stack trace similarity. It consists of a biLSTM encoder and two fully connected layers with ReLU activation. First, stack traces are trimmed and tokenized, and feature vectors are constructed for pairs of similar/dissimilar stack traces. These feature vectors are then fed into S3M for training. Experimental results showed an RR@10 of 0.96 for JetBrains and 0.76 for Netbeans. Shi et al. introduced Abaci-finder [35], which focuses on kernel bug reports. It uses preset regular expressions to extract and trim stack traces. Then, the kstack2vec method is employed to vectorize the stack traces, extracting semantic and kernel-related bias information, key frames, and their context. Finally, an attention-based BiLSTM is used to classify multiple traces. Experimental results demonstrated that Abaci-finder achieved an F1 score of 0.83, outperforming models such as BiLSTM and TF-IDF.

### *4.2. Methods Based on Analysis Coverage*

There are also approaches based on runtime coverage to determine if a test case is a duplicate. As shown in Figure 13, typical fuzzing tools maintain a seed queue during runtime. When the target program requires input, the fuzzing tool retrieves seeds from the queue according to certain rules. The tool also maintains several bitmaps to record the execution paths of the seeds. During runtime, the coverage is evaluated by considering new coverage information, and it is used to determine if a bug is non-duplicate. These methods utilize runtime boundary coverage information to evaluate the level of test duplication. The coverage range during program execution and the frequency of visiting each boundary coverage range can be obtained through instrumentation. When a bug occurs and covers a previously unidentified path or does not match the boundaries covered by previous bugs, it can be considered a non-duplicate crash.

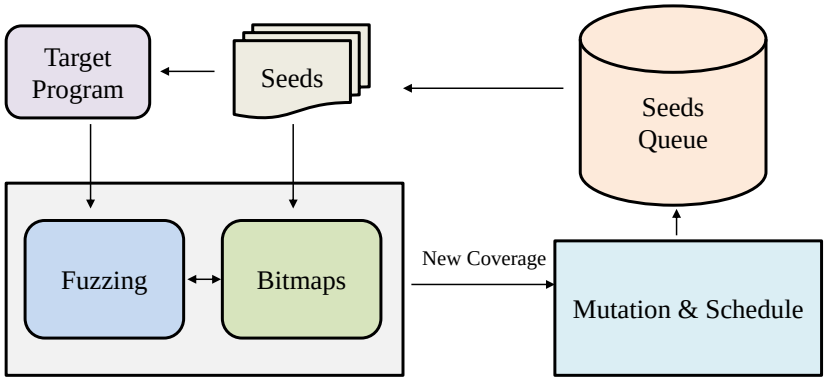


**Figure 13.** Seed processing in fuzz testing.

Figure 14 illustrates the discovery of new coverage information during runtime using AFL as an example. It shows two scenarios: discovering a new path and detecting a change in the execution count of a path. AFL defines corresponding data structures, such as trace_bits and virgin_bits, to determine the presence or absence of a path.

Yeh et al. proposed CRAXTriage [42], a triage tool that utilizes binary code coverage information. It compares the coverage paths of successful executions with the execution path that triggers a bug, aiming to eliminate unnecessary execution paths that contribute to the bug. This approach helps in comparing coverage paths to determine their similarity.

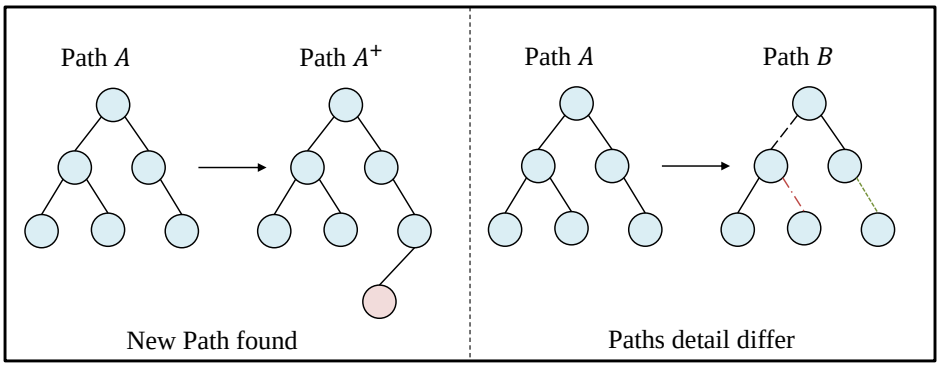


**Figure 14.** New coverage cases.

### *4.3. Methods Based on Context Comparison*

Some studies utilize bug context to determine duplication, which encompasses various types of information, such as control flow graphs, data flow graphs, and more. For example, in one study, RESTful API exception handling, the request and response parameters were extracted as signatures. The Sørensen–Dice coefficient was then used as a distance metric to classify them [43]. The most commonly used method is taint analysis, which involves analyzing the code to generate a control flow graph (CFG), as shown in Figure 15. Symbolic execution tools like Klee are then used for data flow analysis and reconstruction to recover as much related bug context as possible. However, this approach tends to generate a large amount of metadata, which often need to be optimized and condensed. Bug context typically requires multiple types of metadata to accurately describe bug features, thereby enhancing triage accuracy. For instance, relying solely on control flow graphs may lead to misjudgment of crash sites related to transfer control. This is because the source value that caused the transfer control error may not be explicitly represented in the control flow graph. Therefore, the support of data flow analysis is necessary to capture such information.

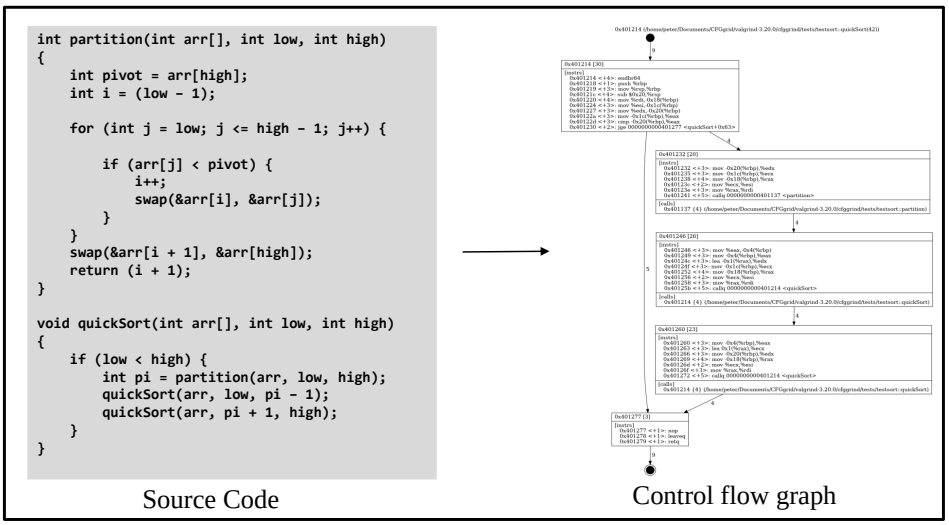


**Figure 15.** Source code converted to CFG.

Based on generated context such as CFG, the uniqueness of a bug can be determined by calculating the similarity of graphs. Currently, kernel methods used for graph similarity computation often employ pattern-matching techniques. Taking the Weisfeiler–Lehman subtree kernel algorithm as an example, as shown in Figure 16, for each node in each subgraph, a first-order breadth-first search (BFS) is performed to obtain its neighboring node set, which is then sorted. Each node is combined with its label and the labels of its neighboring nodes to form a new label, which is then hashed. Finally, the original labels

and the new labels are combined to form feature vectors for comparison and calculation. It can be seen that this method has relatively high computational complexity.

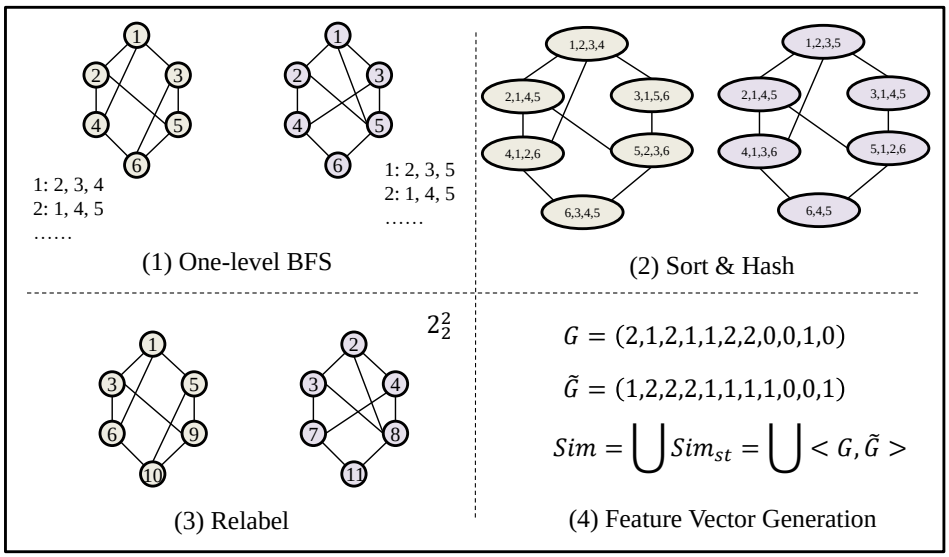


**Figure 16.** Weisfeiler–Lehman subtree kernel algorithm.

Peng et al. studied the features of use-after-free (UAF) bugs and identified two similarities [44]: the creation and deallocation related to the bugs are similar, and the crash contexts are similar. This work mapped the bug context to a 2D plane for clustering and filtering of duplicate bugs. The average clustering time was reduced to 12.2 s. Huisman et al. proposed a clustering method based on symbolic analysis that also provides a semantic analysis of bugs [45]. This work utilized the Klee symbolic analysis tool and employed a clustering-aware search strategy (CLS) to traverse the semantic execution paths of bug-triggering samples and identify the first differing branch from the semantic execution path of successful samples with the longest matching path. This was considered the bug cause and used as a feature for clustering. Experimental results showed that compared to methods based on stack trace and crash site, approximately 50% of test cases could be clustered more finely. Moroo et al. performed bug triage using Rebucket and Party-Crasher [46]. Given a categorized bug pool and a bug to be classified, this work first designed a search engine based on Camel to retrieve M similar bugs (measured using TF-IDF) from the bug pool. Then, using Rebucket, the similarity between the bug to be classified and the M similar bugs was calculated, and the bugs were reordered. Following the idea of Party-Crasher, if the most similar similarity exceeded a threshold, they were grouped; otherwise, they formed an independent class. The experimental results showed an accuracy of approximately 70%.

Cui et al. proposed RETracer [47], which applies reverse taint analysis to binary code. It assumes that the first function containing a memory error is the corresponding bug function, and this forms the basis for bug triage. RETracer does not require complete memory dump information and only needs CPU context and stack information when the bug occurs. It combines forward analysis to address the issue of unrecoverable values. Different handling methods were implemented for tainted analysis within a single function block and across multiple function blocks. Ultimately, a reverse data flow graph was constructed to identify the blamed function. RETracer successfully located 118 out of 140 errors. Jeon et al. introduced CrashFilter [48], which reconstructs the possible path from the crash site to the cause site based on the runtime information of the bug. During construction, reaching-definition analysis was performed, followed by the creation of a definition use Chain. Finally, an exploitability check was conducted, optimizing the process using memory location analysis based on the BinNavi MonoRein framework. Compared to !exploitable, CrashFilter provided more precise evaluations of crash exploitability. Cui et al. proposed REPT [49], a method that utilizes hardware tracing to record control flow and

combines it with core dump to recover data flow. Reverse debugging and taint analysis were performed afterward. To address irreversibility in certain instructions, REPT uses forward execution to recover values. Error correction was employed to handle write instructions with unknown specific addresses. By limiting concurrent writes, the quality of recovered stored values was further improved. The results showed that REPT achieved an average accuracy of 92%, with bug analysis time not exceeding 20 s.

Xu et al. constructed the data flow leading to bugs and performed taint analysis to locate critical statements related to the bug's cause [50]. Although this work does not directly aim at bug deduplication and triage, the identification of critical instructions can still assist in crash localization. POMP builds use-define chains based on control flow information and uses hypothesis testing with memory validation techniques to determine potential values in cases of constraint conflicts, thus recovering data flow. Reverse analysis is then applied to determine the bug cause. Experimental results show that out of 31 tested bugs, the causes of 29 bugs were accurately identified. Mu et al. implemented POMP++ based on POMP [51]. Before conducting reverse analysis, POMP++ enhanced bias analysis by incorporating value-set analysis (VSA) and hypothesis verification, allowing for the recovery of more detailed data flow and increasing the efficiency of bug-cause identification. Experimental results indicated that compared to POMP, POMP++ can recover an additional 12% of data flow while improving efficiency by 60%. Jiang et al. proposed IgorFuzz [52], a technique for crash deduplication through root-cause clustering. It focuses on the proof of concept (POC) that triggers crashes and uses coverage-guided fuzzing to reduce the execution paths while ensuring the triggering of the same bug. This approach overcomes the issue of high errors in comparing sequential execution traces. The control flow graph is used for similarity calculation, and bug clustering is performed using spectral clustering to determine whether a bug can be grouped with existing POCs. The results showed that compared to other approaches, IgorFuzz achieved the highest F score in 90% of cases.

Tonder et al. proposed semantic crash bucketing [53], which utilizes approximate patching techniques, such as automatic patch template generation and rule-based patch application, to determine the root cause of a bug and map it to the corresponding bug, completing bug triage. Experimental results demonstrated that 19 out of 21 bugs were correctly classified using this approach. In 2022, Zhang et al. introduced DeFault [54]. To eliminate the need for root cause analysis, they defined and quantified bug relevance based on the concept of mutual information. Bug classification was performed by determining whether a basic block is present in the bug execution trace. Bug relevance can be used to optimize bug trace analysis and, in turn, help precisely identify the bug cause. Joshy et al. proposed a method for bug deduplication using bug signatures [55]. Bug signatures are composed of the key instructions that trigger a bug. They combine the use of PIN, srcML, and bear to capture dynamic runtime and variable information, compilation information, etc. They use C-Reduce to reduce the instructions and generate bug signatures. When two bugs have signatures that are subsets of each other, they are considered to belong to the same class. Additionally, the stack trace information in bug signatures is used to determine the similarity of the bug signatures themselves for final classification. Experimental results showed an accuracy of 99.1% for bug classification.

## 5. Works Based on Bug Reports

Throughout the entire life cycle of bug reports, in this paper, we focus on the deduplication and triage stages. Essentially, both deduplication and triage aim to find similarities among bug reports. Common methods include text analysis, information retrieval, and various machine learning techniques. For example, Yang et al. combined TF-IDF vectors, word-embedding vectors, and component information from bug reports to comprehensively assess the similarity between reports. The commonly used methods for assessing similarity are briefly introduced in Section 3.2. During deduplication and triage, bug reports are first preprocessed to eliminate irrelevant information. Common preprocessing methods include word segmentation, which involves standard transformations such as

removing punctuation and converting letter cases. Stop word removal is performed to remove insignificant words such as conjunctions and adverbs. Stemming is also applied, which involves extracting words in their base form, regardless of their different expressions.

### *5.1. Information Retrieval Approaches for Deduplication and Triage*

Some work uses IR techniques to enhance the ability to automatically flag bugs. For example, the work of Alawneh et al. [56] can enhance the quality of the bug report by marking the valuable terms in the bug report. Test results showed that the marking accuracy exceeds 70%, which is very beneficial to the deduplication and triage of the bug. Figure 17 illustrates the process of information-retrieval-based approaches in bug triage. This type of work commonly begins by applying preset rules to preprocess bug reports. These rules include removal of stop words and topic-irrelevant words, splitting and tokenization of the contents, and normalization of the text. This preprocessing step aims to clean and prepare the bug reports for further analysis [57]. After preprocessing, the next step involves extracting useful knowledge from various information sources, such as bug properties, stack traces, and others. These sources provide additional context and relevant information about the bugs. Feature extraction techniques are then applied to capture the essential characteristics or patterns from the extracted information. Finally, similarity measures such as the Jaccard coefficient, cosine distance, or other similarity metrics are used to assess the similarity or correlation between bug reports. These measures compare the feature vectors representing the bug reports and determine their similarity or relatedness.

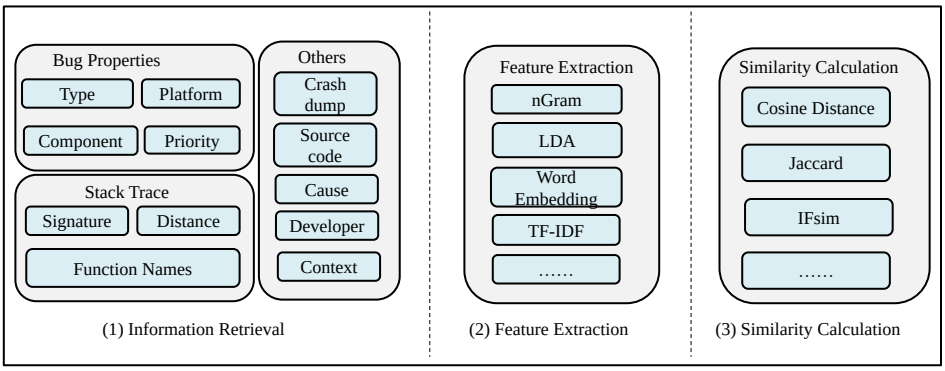


**Figure 17.** General process of information-retrieval-based work.

Prifti et al. found that duplicate bug reports exhibit a certain degree of temporal locality, which can be leveraged to reduce the search space of duplicate reports and improve the search functionality of bug tracking systems (BTS) [23]. Banerjee et al. proposed FactorLCS [21], which suggests that if two bug reports share the longest common subsequence of frequently occurring words in the same order, they are more likely to be similar. Based on this idea, they detected duplicate bug reports by matching the sequences. The authors introduced match size within group weight (MSWGW) to weigh the scores of the longest common subsequence (LCS) and obtain the final matching score, aiming to reduce the rate of false matches. Banerjee et al. proposed a fusion framework based on multilabel classification to categorize bug reports into different groups [22]. The authors trained a multilabel classification model using the MULAN module in the Weka machine learning software package. For a given bug report to be classified, the framework retrieved the top 20 potential similar reports by combining the highest scores from multiple labels. Lee et al. proposed a time-based model based on BM25Fext to model the submission time and version information of bug reports [58]. This model combined the textual information, category information, and time features to rank potentially similar bug reports. Wang et al. proposed an approach to improve bug management using correlations in crash reports [59]. They determined that if the same bug corresponds to different crashes under different scenarios, these crashes can be correlated. They introduced five rules to determine whether crashes

are correlated, including three based on stack trace signatures, one based on temporal locality, and one based on the textual similarity of crash comments. They then used crash correlation to identify duplicate bugs. Experimental results showed that the deduplication method achieved a recall rate of 50% for the Firefox dataset and 47% for the Eclipse dataset, with precision rates of 55% and 35%, respectively.

Rakha et al. conducted experiments and found that the effectiveness of previous bug report duplication detection methods was overestimated by 17% to 42%. As a result, they suggested using the resolution field extracted from bug reports (e.g., the "FIXED" or "WONTFIX" tags) to improve the efficiency of duplication detection [60]. Banerjee et al. developed a system for evaluating the similarity of documents [61]. Unlike previous works that assumed reports to be duplicated, this system assigns an unknown status to newly inputted reports. It then analyzes the cosine similarity of the text, time windows, and document factors, comparing them with the reports in the repository. For reports identified as duplicates, the system listed 20 potential matching reports. Finally, a Random Forest model was employed for classification purposes. Savidov et al. presented Casr [62], a bug classification approach based on two parameters: the distance of a function from the top of the stack and the relative distance between identical function calls within two call stacks. Smaller parameter values indicate higher similarity. Experimental results demonstrated that this method effectively clustered similar crash reports. Saber et al. proposed DURFEX [39], which determines whether bug reports are duplicates based on stack traces. This work involved extracting features from stack traces in multiple steps. First, function names were replaced with package numbers to reduce the number of features. Then, N-grams were used to construct feature vectors. The similarity was measured by calculating the distance between feature vectors and vectors in the model. Finally, similarity and component features were considered together to generate a list of potential duplicates. Experimental results showed that using two-gram sequences achieved a recall rate of 86%, outperforming the use of distinct function names only (recall rate of 81%) and reducing execution time by 70%.

Budhiraja et al. proposed the LWE method [63], which combines latent Dirichlet allocation (LDA) and word embedding for deduplication. LWE leverages the high recall rate of LDA and the high precision of word embedding. It first modifies bug reports using LDA to exclude obviously dissimilar reports, then uses a word-embedding model to determine the top k most similar reports. Experimental results showed a recall rate of 0.558 for the top 20 reports. Mu et al. analyzed and summarized the possible causes of duplication [6], including input differences, thread interleaving, memory dynamics, different kernel versions, inline functions, and sanitizers. They designed targeted deduplication strategies based on these causes, such as using stable versions of the kernel, swapping the execution of proofs of concept, reducing the impact of thread interleaving and inline functions using stack traces, selecting specific sanitizers, and replacing the slab allocator with the slub allocator to mitigate the impact of memory dynamics. Experimental results showed the effective identification of duplicate error report pairs with a true-positive rate of 80% and a false-positive rate of 0.01%. Chaparro et al. proposed three methods for querying duplicate bug reports, which are based on bug title (BT), observed behavior (OB), and a combination of BT and OB [64]. Karasov et al. proposed an approach based on stack trace information [65]. They compared the similarity between a given stack trace and existing grouped stack traces, taking into account the time stamps of stack trace occurrences. They used a linear aggregation model to calculate similarity rankings and selected the most similar group for insertion. Experimental results demonstrated a 15% improvement in RR@1 compared to the baseline.

James et al. introduced CrashSearch [66], a method for effectively mapping newly discovered bugs to categories in a bug dataset. They extracted features from bug reports, such as bug type and crash function, and generated bug fingerprints using the MinHashing algorithm with MurmurHash. In the similarity determination process, bugs were divided into multiple bands, and locality-sensitive hashing (LSH) was used to compute the hash

values for each band. By comparing the bug index with the index of the bug band, similar bugs were stored in the same band, thus facilitating bug triage. Additionally, this work used the bidirectional extension algorithm to further classify bugs based on the different relationships between similar bug pairs. Experimental results showed that CrashSearch achieved improved F scores compared to Tracesim, top-k prefix match, major hashing, and minor hashing, with increases of 11%, 15%, 19%, and 31%, respectively. Yang et al. proposed a custom knowledge-based detector (K-detector) [67]. This method takes source code, crash dumps, and historical crash data as input. It first performs data filtering to extract information such as functions and calls from the stack trace. Then, using the the AST generated by Clang, it establishes correspondences between functions and components and defines a mathematical model to measure the similarity between bugs. Experimental results showed that bug report classification for SAP HANA achieved an AUC of 0.986.

Dhaliwal et al. performed a two-level classification of bugs based on stack traces [32]. The first level was based on the top-method signature, and the second level used the Levenshtein distance between traces to determine similarity. Experimental results showed that bug fix time was reduced by 5% after grouping bugs according to method. Park et al. proposed CosTriage [68], which determines whether assigning a bug to a specific developer has a low enough cost based on the similarity of bug content and the collaboration relationship with the developer. CosTriage aims to not only triage bugs to the appropriate individuals but also minimize costs. It categorizes bugs, models the developers themselves, and captures attribute changes over time. It then calculates cost scores and ranks them, recommending the developer with the lowest cost. Experimental results showed that CosTriage helped reduce costs by 30%.

Hindle et al. used context features to optimize bug deduplication based on information retrieval (IR) [69]. These features included software architecture, non-functional descriptions (such as portability and reliability), topic words, random content, and more. This work first extracted error-related information from bug reports, then measured their similarity to an existing database using BM25F. Additionally, a machine-learning-based classifier was used to identify duplicated bug reports in the dataset. The results showed an improvement in the accuracy of duplicate bug detection of up to 11.5%, a 41% increase in Kappa measurement, and a 16.8% increase in AUC measurement. Badashian defined the bug triage problem as the allocation of a group of bugs to a group of developers with the lowest cost, considering both minimization of tossing and the cost of developers [70]. This work first modeled developers based on their contributions to code and Q & A platforms. They constructed a professional knowledge graph and a knowledge domain graph required for bug reports. Then, they used the Jaccard similarity measurement to find the developer whose professional knowledge graph was most similar to the knowledge domain graph of the bug report. They considered using evolutionary algorithms or the Kuhn–Munkres algorithm for an optimal solution to the cost of developers for bug resolution.

Zhang et al. considered the impact of developers playing different roles during the bug-fixing process [71]. They first used the Unigram model to abstractly represent bug reports, then used Kullback–Leibler (KL) divergence to measure the similarity between bug reports. This allowed them to extract relevant feature information about potential fixers from similar bug reports in the dataset. Finally, they ranked the potential fixers based on the extracted feature information to complete the classification. Experimental results showed that the average precision, recall rate, and F1 measure for recommending 10 developers were approximately 75%, 40%, and 52%, respectively. Xia et al. extended latent Dirichlet allocation (LDA) and proposed a multifeature topic model (MTM) [25]. They designed TopicMinerMTM to compute the likelihood of recommending a developer. The process involved two stages: model construction and recommendation. In the model construction stage, MTM was used to extract multiple topic feature vectors from bug reports, which were then input to TopicMinerMTM for model construction. In the recommendation stage, TopicMinerMTM was used to score and recommend developers. After the recommendation, the model was updated using the recommended results. Experimental results showed that

TopicMinerMTM achieved an average top-one precision of 68.7% and a top-five precision of 90.8%.

Goyal et al. proposed three models [72]. The first one is the "Visheshagya" time-based bug triage model, which considers the factor of developers' knowledge changing over time. Experimental results showed a 15% improvement in accuracy compared to models that do not consider the time factor. The second model is called "W8Prioritizer", which assigns different weights and priorities to bug parameters to help classify bugs. Experimental results demonstrated a 29% increase in accuracy compared to models that did not consider the priority factor. The third model, "NRFixer", was developed specifically for non-reproducible bugs. It uses bug metadata to predict the probability of an NR (non-reproducible) bug being fixed. The evaluation results indicated that NRFixer achieved an accuracy of around 70%. Zhang et al. proposed En-LDA [73], which utilizes LDA (latent Dirichlet allocation) to extract topics from bug reports. The topics are used as word probability labels, and the entropy of each word is calculated and aggregated based on the topic distribution to optimize the number of topics. Experimental results showed that En-LDA achieved an RR@5 of 84% for JDT and an RR@7 of 58% for Firefox. Pham et al. focused on bug classification using the symbolic analysis tool Klee [45]. They searched for execution paths that caused bugs through a combination of symbolic analysis and a clustering-aware search strategy. By comparing the execution paths with successful paths using longest common prefix (LCP), they analyzed the root cause of bugs and classified them or created new ones. The experimental results showed that more fine-grained classification results could be obtained using this approach.

Hindle et al. proposed a method for bug report duplication detection called continuous querying [74], which involves continuously searching and querying bug reports to determine if they could be duplicates before reporting the bugs. Experimental results showed that this query-intensive method can prevent over 42% of observed duplicate bug reports from occurring. Zhao et al. addressed the issue of different formats between lightweight bug reports from third-party testing and general bug reports [10]. They proposed a unified bug report triage framework that involves feature extraction using information gain (IG) and chi-square (CHI) statistical methods. They then adjusted parameters, vectorized the features using TF-IDF, generated LDA representations for reports, and applied a three-layer neural network combined with SVM for triage. Experimental results showed that the proposed framework achieved an optimal accuracy rate of 49.22% for Eclipse, 85.99% for baiduinput, and 74.89% for the Mooctest dataset. Yadav et al. developed a scoring system for developers based on priority, versatility, and average fix time [75]. They used similarity measurement methods such as Jaccard and cosine similarity to identify bugs in the database that were similar to the current bug. This helped determine potential developers who could handle the bug and rank them based on their respective scores. The experimental results showed average accuracy, precision, recall rate, and F score of 89.49%, 89.53%, 89.42%, and 89.49%, respectively.

Alazzam et al. proposed a method called RSFH (relevance and similarity-based feature hierarchy) for bug triage [76]. It involves treating the key terms in bug reports as nodes in a graph. Feature extraction is performed using latent Dirichlet allocation (LDA) to extract topics. The neighborhood overlap of nodes is used to enhance the feature analysis of bug reports. Finally, graph classification is used for triage. Experimental results showed improvements in accuracy, precision, recall, and F measure compared to term frequency inverse document frequency (RFSTF) and CNN methods when classifying bugs of different priorities. For example, for bugs with priority p1, the accuracy, precision, recall, and F measure were 0.732, 0.871, 0.732, and 0.796, respectively. Neysiani et al. proposed a feature extraction model to aid in bug triage deduplication [12]. The model aggregates various features extracted from bug reports, including multiple text features extracted using TF-IDF, time features, context features, and classification features. The authors also proposed a heuristic feature efficiency detection method for evaluation. Experimental results showed that using the extracted features improved deduplication accuracy, recall

rate, and F measure by 2%, 4.5%, and 5.9%, respectively. Nath et al. addressed discrete and non-discrete features in bug reports using different approaches [77]. They employed principal component analysis (PCA) for discrete features, specifically using the one-hot encoded method for analysis. For non-discrete features, they transformed the text into a bag-of-words (BOW) representation and used an entropy-based keyword extraction method (greedy variant) for analysis, defining an entropy threshold to filter features. They combined developer contributions and used multiple classifiers to calculate the probability of assigning a bug to a specific developer or team. Experimental results showed that random forest achieved an accuracy of 79% for top-1, 87% for top-5, and 90% for top-10 team assignments. For developer assignments, the accuracy was 54% for top-1, 63% for top-5, and 67% for top-10.

Panda et al. pointed out that many bug reports contain ambiguous descriptions and excessive terminology [27], leading to classification hesitation. Traditional machine learning or precise information retrieval methods may not be effective in such cases. Therefore, they proposed an approach based on intuitionistic fuzzy sets (IFS) for classification in the presence of uncertainty. The work included two subparts: IFSDTR and IFSDCR. IFSDTR focuses on the relationship between developers and terminology, while IFSDCR aims to assign multiple fixers to multiple bugs. Experimental results showed that the accuracy of the IFSDTR technique was 0.90, 0.89, and 0.87 for the Eclipse, Mozilla, and NetBeans datasets, respectively. The accuracy of IFSDCR was even higher for the Eclipse, Mozilla, and NetBeans datasets, reaching 0.93, 0.90, and 0.88, respectively. Krasniqi et al. proposed a quality-based classifier that categorizes bugs into six types based on the ISO 25010 standard [57], including reliability, maintainability, usability, and others. They combined multiple feature extraction tools, such as TF-IDF, chi-square, and random trees. Experimental results showed that using TF-IDF + chi-square with random forest yielded the best results in terms of accuracy (76%), recall (70%), and F1 score (70%) for the triage of bugs related to quality. However, the performance was not as good when triaging bugs related to functionality. Panda et al. proposed using fuzzy logic, specifically intuitionistic fuzzy sets (IFS), for bug triage [26]. They first applied LDA to the bugs to generate topic models and classify them into several classes. They then used the Sugeno complement generator to estimate the association value between developers and bug models. Finally, they used IFSim to calculate the similarity between bugs and developers. Experimental results for the Eclipse dataset showed an accuracy of 0.894, precision of 0.897, recall of 0.893, and F measure of 0.896.

Khanna et al. proposed TM-FBT (topic modeling-based fuzzy bug triaging) [78], which combines topic modeling and fuzzy logic for bug triage. Topic modeling is used to define bug-related models, and fuzzy logic is used to learn the relationship between developers and bug-related models, enabling the mapping between developers and bugs. Experimental results on the Eclipse, Mozilla, and NetBeans datasets showed accuracy rates of 0.903, 0.887, and 0.851, respectively, for the TM-FBT method. Wu et al. introduced CTEDB [79], a method that utilizes information retrieval (IR) techniques to detect duplicated bug reports. This approach begins by extracting terminology from bug reports using Word2Vec and TextRank. Then, it computes semantic similarity using Word2Vec and SBERT. Finally, it employs DeBERTaV3 to process the extracted terminology from bug reports and calculate the confidence score for duplicate detection.

Some works focus on data reduction for bugs before performing triage, which involves converting bugs into important features stored in a bug repository. This approach has the advantage of reducing the dimensionality of bugs, reducing redundancy, and improving the quality of the bug dataset. Priyanka et al. surveyed the literature on bug triage based on data reduction and described the methodological framework, including preprocessing, vector model construction, data reduction, feature extraction, and triage modules.

### *5.2. Machine Learning Approaches for Deduplication and Triage*

Based on the machine learning approach, a class of methods often draws inspiration from NLP techniques. Figure 18 illustrates the process of using machine learning for deduplication and triage. Unlike IR-based approaches, these methods typically take the textual information directly from bug reports, developer information, and stack traces as input. They then employ suitable feature extraction methods such as LDA, graph extractors, etc., to extract feature vectors or graph representations that can be used by machine learning models. Finally, trained machine learning models such as CNN, LSTM, and DNN or classical ML models like SVM are utilized for deduplication and triage tasks.

Ebrahimi et al. proposed a method for detecting duplicate bug reports based on stack traces and hidden Markov models (HMMs) [80]. They divided the stack traces into several groups, each containing a main stack trace and multiple stack traces marked as duplicates. They trained an HMM model to compare and determine whether a new stack trace is a duplicate. Experimental results showed that for the Firefox and GNOME datasets, the mean average accuracy reached 76.5% and 73%, respectively. When k > 10, the RR@K exceeded 90%. Rodrigues et al. introduced a DNN model for bug deduplication [81]. They first used a ranking method to identify the k most similar candidates to the target report. Then, they employed a soft attention alignment approach to compare the content of the reports and determine if they were duplicates. The soft alignment model consisted of a categorical module and a textual module. The categorical module predicted the probability of similarity between the target report and previous reports, while the textual module dynamically extracted information from the textual content to remove duplicated reports. Experimental results showed an improvement of approximately 5% in the recall rate across four datasets. He et al. proposed a method based on a dual-channel CNN to detect duplication in bug reports [82]. The key of this approach was to use word2vec to transform bug reports into two-dimensional matrices and combine the matrices of two bug reports into a dual-channel matrix. They then trained a CNN to predict the similarity of the dual-channel matrices extracted from the input bug reports. Experimental results showed that the detection accuracy, recall rate, precision, and F1 score all exceeded 0.95 on datasets such as Open Office.

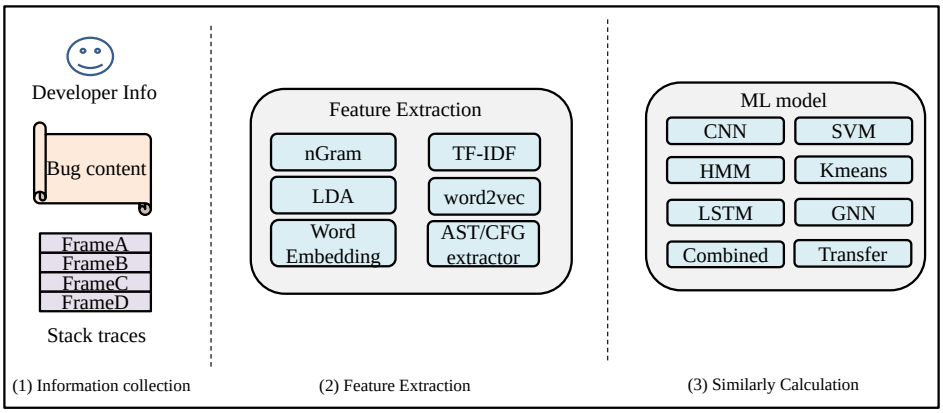


**Figure 18.** General process of machine-learning-based work.

Aggarwal et al. presented a method that uses the BM25F similarity measure to classify bug reports by extracting the generic and project-independent context of the bug report and word lists derived from software engineering textbooks and multiple open-source projects [83]. They performed the classification on various machine learning models and achieved a classification accuracy of approximately 92%. Angell et al. focused on triaging hardware bugs [4]. They used bug reports and hardware design files (HDFs) as inputs. First, they parsed the HDF to generate several structure-related subgraphs to ensure the inclusion of potential problematic signals. Bug reports were utilized to extract signal values, which were used as features and input to a k-means model for clustering.

Experimental results showed that the average verification efficiency increased by 243%, with a 99% confidence interval. Dedik et al. aimed to test the differences between bug triage requirements in industrial settings and open-source projects [84]. They employed an SVM + TF-IDF method for classification on a private company dataset. Experimental results showed an accuracy of 53%, precision of 59%, and recall of 47%. These results are similar to the features exhibited in open-source projects. Lin et al. defined duplication detection as a ranking problem [85]. They used TF-IDF to compute term weights and BM25 and Word2Vec to calculate similarities and trained an enhanced SVM model (SVM-SBCTC) to detect duplicated bug reports. Experimental results demonstrated an RR@5 improvement of 2.79% to 28.97% compared to SVM-54.

Lee et al. used a combination of CNN and Word2Vec for bug triage [86]. They transformed bug reports into feature vectors using Word2Vec, taking into consideration the handling of multilingual environments and field-related terminology. A CNN was then employed to compute the probability of assigning a bug report to a specific developer. Experimental results showed that compared to manual triage, their approach achieved 82.83% and 35.83% higher performance in top-one and top-three accuracy than open-source projects. Xuan et al. attempted to address the issue of insufficient labeled bug reports [87]. They used an expectation maximization enhanced naive Bayes classifier to handle a mixture of labeled and unlabeled bug reports. The labeled bug reports were used to train the classifier, and based on this, an iterative process of labeling the unlabeled bug reports (E step) and retraining the classifier (M step) was performed. The training process incorporated a weighted recommendation list to facilitate iterative training of the classifier. Experimental results demonstrated that this semisupervised method achieved a 6% improvement in accuracy.

Song et al. proposed DeepTriage [88], a method that combines bidirectional LSTM with pooling for bug triage. DeepTriage considers both the textual information in bug reports and the activity information of developers. It consists of four layers. The input layer encodes the textual content of the bug report and the developer sequence, which is then passed to the feature extraction layer, which contains two extraction models. Bidirectional LSTM is used to extract features from the bug report content, while unidirectional LSTM is used to extract features from the developer activity sequence. Experimental results showed a top-one accuracy of 42.96%. Chaparro aimed to improve the quality of textual information in bug reports to enhance the accuracy of IR-based deduplication [89]. First, the author defined and identified relevant information about the observed behavior (OB), expected behavior (EB), and steps to reproduce (S2R) in bug reports. Then, heuristic text phrasing or machine learning models like SVM were used to predict whether the bug reports contained OB, EB, or S2R, and recommendations were provided to the reporter to improve the related descriptions. Finally, the improved OB, EB, and S2R were utilized to optimize duplication detection. Xi et al. presented an effective approach for routing of bug reports to the right fixers called SeqTriage [90], which considers the potential relationship chain between developers and fixers. SeqTriage consists of an encoder that extracts text information into hidden states/features and a decoder that computes the tossing sequence. The encoder is based on bidirectional RNN and GRU neurons, while the decoder uses RNN to model the tossing sequence. An attention model is introduced in the decoder to account for the varying contribution weights of each word to the overall text. In practical applications, the output could be the last developer in the tossing sequence. Experimental results showed that SeqTriage outperformed other approaches by 5% to 20% in terms of accuracy.

Xie et al. proposed DBR-CNN [91], which preprocesses bug reports by removing irrelevant words and tokenizing them. The tokenized words are then transformed into feature vectors using word embedding. Finally, a CNN is used to compute the similarity score between two bug reports. Experimental results showed that the F score and accuracy reached 0.903 and 0.919, respectively. Jiang et al. introduced the TERFUR framework to address fuzzy clustering test reports (FULTERs) [24]. This framework clusters bug reports

by defining rules to identify and remove invalid reports. The bug reports are preprocessed by removing stop words and enhancing the textual content using NLP models. The vector space model is employed to calculate the similarity between bug reports, and a merging algorithm is used for clustering, achieving an average accuracy, recall rate, and F1 measure of 78.15%, 78.41%, and 75.82%, respectively. Alenezi et al. extracted structural features from bug reports [92], including component, operating system, and priority. They used a naive Bayes classifier to predict the probability that a developer can handle a bug. Experimental results showed F scores of 0.633, 0.584, and 0.38 for Netbeans, Freedesktop, and Firefox, respectively.

Budhiraja et al. employed word embedding to convert bug report descriptions into vectors and trained a DNN model called DWEN for classification [93]. Experimental results showed an RR@20 (reciprocal rank at 20) score of over 0.7. Choquette-Choo et al. considered the role of developers in bug triage and proposed a DNN model to predict triage results [29]. The model had two outputs: team prediction and more refined developer prediction. It utilized a multilabel classifier and was trained in two stages. The first stage focused on training for team bug resolution labels, while the second stage involved training for labels related to the relationships between developers, coding teams, and bugs. Experimental results showed a 13% improvement in 11-fold incremental-learning cross-validation accuracy (IL-CV) compared to traditional DNN models. The accuracy for team assignment reached 76%, and for individual assignment, it reached 55%. Kukkar et al. proposed a CNN- and boosting-enhanced random forest (BCR) structure to classify bugs according to the severity of bug reports [94]. They first preprocessed the bug reports by removing irrelevant information and extracting text features using N-gram. These features were then fed into a CNN to extract bug severity features. Finally, random forest was used to categorize each bug, making the result more reliable and avoiding overfitting issues. The DNN model achieved an accuracy of 96.34% and an F measure of 96.43%. Xi et al. presented iTriage [95], which extracts features for bug triage based on the textual content, metadata, and tossing sequence of bug reports. The textual content and tossing sequence are used to train a feature learning model, while the metadata are used to train a fixer recommendation model. The feature learning model takes the textual content as input, incorporates attention mechanisms to extract word properties, and adapts them to corresponding developers, generating a tossing sequence combined with metadata for subsequent classification tasks using a simple neural network. Experimental results showed that iTriage achieved a 9.39% improvement in top-one accuracy compared to TopicMinerMTM.

Mani et al. proposed DeepTriage [96], which utilizes a deep bidirectional RNN and attention models to identify and preserve important content in bug reports. After extracting features, a softmax classifier is used for classification. Experimental results showed a rank-10 triage accuracy ranging from 34% to 47%. Catolino et al. believed that mining the root cause of bugs can help with classification [97]. They manually labeled 1280 bug reports based on the root cause to determine potential causes. They then profiled the causes based on their frequency, associated topics (extracted using enhanced LDA: LDA-GA), and required fixing time. Finally, they proposed a classification model based on logistic regression. Experimental results showed an F measure of 64%. Poddar et al. presented a neural network architecture that can simultaneously perform deduplication and clustering [98]. Bug reports are first transformed into d-dimensional vectors using word embedding, and bidirectional GRU units generate two sets of g-dimensional vectors containing coarse-grained topic information and other fine-grained information. The model employs topic clustering using vector topic information combined with self-attention, and the remaining information is utilized for deduplication using conditional attention. Experimental results showed F1 scores of 0.95 and 0.88 for deduplication in Firefox and JDT, respectively. Sarkar et al. used machine learning models to classify bug reports based on Ericsson's bug data [99]. They extracted text features from bug reports using the TF-IDF method, obtained categorical features using one-hot encoding, and collected log features such as alarms and crash dumps. Bug triage was performed using L2-regularized logistic

regression with the Liblinear solver. Experimental results showed precision and recall rates of 89.75% and 90.17%, respectively, with a confidence level of 90%.

Pahins et al. proposed T-REC [100], which combines machine learning and information retrieval (IR) techniques. After preprocessing and normalizing text information, unstructured natural language descriptions and structured software engineering data are obtained. The vector space model (VSM) is used to convert text into 300-dimensional vector representations, and machine learning ranking methods are employed to rank the technical groups. T-REC also uses a series of BM25F-based extension versions for detection. Finally, a noisy-or classifier is used to calculate the joint probability distribution and provide the final recommendations. Experimental results showed an accuracy of 76.1% for ACC@5, 83.6% for ACC@10, and 89.7% for ACC@20. Guo et al. used a CNN-based method for bug triage [101]. This approach involved converting the content of bug reports into vector representations using word2vec. By combining developer activity information, a CNN was utilized to map bug reports and developers. Experimental results showed a top-10 accuracy of 0.7489. Lee et al. proposed using backpropagation techniques to improve the accuracy of multiple LDA (latent Dirichlet allocation) [11]. They first applied LDA to bug reports for topic modeling and classification, obtaining the union topic set (UTS). They then performed modeling and classification on priority and severity to obtain the partial topic set (PTS), allowing for the identification of cases where priority or severity was not correctly captured in the classification. Bug reports that were misclassified were analyzed to extract the feature topic set (FTS), and based on FTS information, the bug reports were retriaged to obtain an updated UTS. Experimental results showed that the classification accuracy for different priorities exceeded 84%. Xiao et al. proposed HINDBR, a neural network for detecting semantic similarity in bug reports [102]. It transforms the structured or unstructured semantic relationships in bug reports into low-dimensional vector representations and determines whether reports are duplicates based on the distance between vectors in the latent space.

Zhang et al. argued that directly assigning bug triage to a specific developer in the industry is not reasonable due to frequent personnel changes [103]. Therefore, they proposed using components as the target for triage and recommending the current active developer associated with the component. The process of this method is similar to typical machine learning approaches. Bug reports are preprocessed to remove irrelevant information; then, LDA is used to extract topic features. Finally, a DNN model is constructed to predict the relationship between topic features and corresponding labels. Experimental results showed RR@5 rates of 85.1%, 70.1%, and 92.1% for recommending active developers on the JDT, Platform, and Firefox datasets, respectively. Russo et al. employed an NLP approach using word2vec to transform the text in bug reports into bag-of-words (BOW) and extract feature vectors [104]. LSTM was trained for bug triage. Experimental results showed that this approach achieved approximately 78% accuracy after six iterations. Neysiani et al. proposed a new feature extraction model that uses a combination of TF-IDF-based features to improve the efficiency of deduplication [105]. He et al. used a hierarchical attention network for automatic bug triage [106]. Considering the limitations of existing feature extraction methods, the authors utilized Word2Vec and GloVe to extract features from bug reports. A hierarchical attention network based on an RNN was employed to overcome the limitations of CNNs, focusing on local information and LSTM being distracted by long sequences. The hierarchical nature in this context refers to the use of different attention mechanisms for words and sentences. Experimental results showed accuracy ranging from 50% to 65% for different datasets.

Wang et al. built a hybrid DNN model using RNN and CNN for triage [107]. LSTM was utilized to capture sequence information in the text, while CNN was used to extract local information. Attention mechanisms were employed to weight the features, and the features were fed into fully connected and softmax layers to obtain the triage results. Experimental results showed top-five accuracy of 80% on the Eclipse dataset and 60% on the Mozilla dataset. Yu et al. proposed a bug triage model called BTCSR that considers the co-

operation and order relationships among elements involved in the bug-fixing process [108], including reviewers and fixers. BTCSR first analyzes the content of bug reports, extracts topic and terminology-related features, and searches for similar bug reports in the dataset. Then, a sequence graph model is constructed that considers the cooperation relationships of various elements and includes several types of metapaths extracted from historical bug reports. The most suitable fixer is determined by comparing the similarity of paths. Experimental results showed RR@3, RR@5, and RR@10 of 51.26%, 63.25%, and 74.14%. respectively, on average, outperforming SVM and DeepTriage. Zaidi et al. transformed the bug triage problem into a node classification problem in a graph [109], which can be solved using a graph convolution network (GCN). This method first performs common preprocessing on bug reports, such as stop word removal, then converts the documents into graphs, where words are transformed into nodes and the relationships between words and between words and documents are converted into edges. TF-IDF is used to weight the edges. The transformed graph is then used as input for GCN to compute the classification results. Experimental results showed top-10 accuracy rates of 84%, 72.11%, and 66.5% on the JDT, Platform, and Firefox datasets, respectively.

Jahanshahi et al. proposed DABT [110], a method that considers bug dependency and factors such as developers' fix time. They first built a bug dependency graph (BDG) to represent the existence of dependencies among bugs. They used LDA for topic modeling to estimate the time required for developers to fix bugs. Based on TF-IDF features, they used SVM to obtain classification labels and calculate the suitability between bugs and developers. Finally, they abstracted the triage problem into an integer programming problem to determine how to assign bugs to developers or defer them. Experimental results showed that although DABT had a decrease in triage accuracy, it significantly reduced the time required for bug fixes by less than 50%. Zaidi et al. used one-class SVM to address the issue of new developers who cannot effectively participate in bug triage [111]. They trained an independent model for each developer to determine whether a bug can be assigned to them. The process aligns with the common process of preprocessing bug reports, extracting features, training classifiers, and making predictions. Experimental results showed an average accuracy of approximately 93% and an average recall rate of approximately 53%. Zhang et al. focused on the importance of assigning bugs to less experienced developers to maintain project sustainability [112]. They proposed SusTriage, which categorizes developers into three types—core, active, and peripheral developers—and models each type separately. For each type of developer, they used multimodal learning to construct a model that predicts the probability of using a certain type of developer for a given bug report. Finally, they integrated multiple recommendation models for different types of developers, along with corresponding learning weights, to make recommendations. Experimental results showed that SusTriage outperformed baseline methods, with a 69% increase in MAP and a 61% increase in MRR in the Eclipse project. It also increased MAP by 57% and MRR by 46% in the Mozilla project. For Eclipse and Mozilla, it increased diversity by 14% and 6% and entropy by 55% and 13%, respectively.

Aktaş attempted to classify bugs for the technical team in an industrial environment [113]. They first tokenized the description and summary parts of bug reports and transformed them into n-dimensional vectors using TF-IDF, corresponding to the terminology and weights in the reports. They then trained a two-level classifier, where the lower-level classifiers were a series of classifiers such as KNN, SVM, and decision tree, and the high-level classifier was a linear logistic regression that integrated the predictions of the lower-level classifiers. Experimental results showed a slight drop in accuracy compared to manual classification, with an accuracy of 0.83 compared to 0.86. However, it significantly reduced the human resource cost. Jang et al. extracted features from bug reports and inputted the obtained feature vectors into a CNN+LSTM model for bug triage recommendation [28]. The recommended accuracy was approximately 52.4%. Aung et al. designed a multitask learning model for bug triage [114] that uses a context extractor and an AST extractor to extract textual information and code information from bug reports and obtain developer

and bug categories. They used a context augmenter to replace words in the original reports to enhance functionality and generate augmented reports. Finally, they used a CNN-based text encoder and a BiLSTM-based AST encoder to simultaneously determine the developer to be assigned and the possible bug type. Experimental results showed an average accuracy of 57% for developers and 47% for bug types.

Lee et al. proposed an optimized version of bug triaging called LBT-P [49], which is based on pretrained language models (PLMs). They used the patient knowledge distillation (PKD) technique to transfer knowledge from the PLM module to the Roberta module, making it faster and more efficient for specific tasks while minimizing knowledge loss. To address the catastrophic forgetting problem in transfer learning, this work used knowledge preservation to prevent forgetting of general knowledge. They froze earlier layers, since general knowledge is retained in this part. They defined a weighted loss function based on the presentation of earlier layers to handle the overthinking problem, where excessive computation may lead to performance degradation. The framework consists of a text-embedding module based on PLMs, which converts bug reports into text-embedding vectors, and a classifier based on a CNN (rather than RNN) due to the high time cost. Experimental results showed accuracy rates of 75.2%, 82.7%, 78.2%, and 79.3% on Google Chrome, Mozilla Core, Mozilla Firefox, and a private dataset, respectively, outperforming TF-IDF, DBRNN-A, and RoBERTa-large + CNN. Yu et al. aimed to mitigate the impact of low-quality bug descriptions and proposed MSDBT [115], which also considers component and product information. They primarily used LSTM to process the content of bug reports and calculate the textual content and fixer features. They introduced an attention layer to determine the influence of features on the results and generated a series of probabilities corresponding to fixers for each bug. Experimental results showed RR@3, RR@5, and RR@10 values of 0.5424, 0.6375, and 0.745, respectively.

Wu et al. built upon the developer collaboration network (DCN) and considered the interaction between developers to propose the ST-DGNN model for bug triaging [9]. It includes the joint random walk (JRWalk) mechanism and a graph recurrent convolutional neuron network (GRCNN). JRWalk is used to sample the developer collaboration network and obtain the attributes related to developers themselves and their interactions. GRCNN uses the sampled attributes to learn the spatiotemporal cyclic features of DCN in hour, day, and week units. This functionality was achieved using a CNN (for extraction of spatial features) and LSTM (for extraction of temporal features). An attention layer was introduced to balance the three types of time-spatial features. Finally, a simple neural network was used to perform three classification tasks: (1) determining the most suitable developer to fix a specific type of bug, (2) predicting whether a developer has a good willingness to collaborate, and (3) assigning bugs to the predicted most suitable developer. Experimental results showed an average F1@k value of 0.7 for multiterm prediction in bug triage tasks. Chao et al. introduced DeepCrash [116], which utilizes frame2vec to split and extract frame representations from stack traces. A Bi-LSTM-based DNN model was used to convert the frame representations into vector representations and compute similarities. Finally, the Rebucket clustering algorithm was applied to complete the triage. Experimental results showed that DeepCrash achieved an F score of 80.72%. Zaidi et al. proposed using a transformer (BERT) to recommend developers for a given bug [117]. Structural features were extracted from bug report descriptions, and the unstructured text was tokenized and fed into a fine-tuned BERT model. The output was connected with fully connected layers and a classification layer for the final classification. Fine-tuning was essentially a transfer learning method, and when adding a new developer, 40% of existing developer data were randomly selected for retraining of the model. Experimental results showed that this work achieved over 60% top-10 accuracy on multiple datasets.

Samir et al. emphasized the significance of interpretability in improving the rationale of bug triage [118]. They proposed two interpretable machine learning models for bug triage. The first model is designed to model developers and predict the bug types that are best-suited for a specific developer to fix. The second model is aimed at modeling bugs

and predicting the most suitable developer to fix a particular bug. Chauhan et al. proposed DENATURE, a method that combines information retrieval (IR) and machine learning (ML) techniques for detection of duplicate bug reports and bug triage [30]. They first convert bug reports into TF-IDF vectors using an IR-based approach and calculate cosine similarity to determine if bug reports are duplicates. Then, they utilize machine learning classifiers such as SVM and logistic regression to perform bug triage, which involves identifying the type of bug. Jiang et al. conducted experiments and demonstrated that both IR-based and ML-based methods suffer from information loss when evaluating text similarity [31] and that ML-based approaches do not necessarily outperform IR-based methods. To address this issue, the authors proposed a hybrid approach that combines both IR and ML techniques for calculation of text similarity. In this approach, IR methods are used to compute textual and semantic features, while ML methods are employed for classification and prediction tasks. By integrating these two approaches, the authors aimed to improve the accuracy and performance of text similarity evaluation and bug triage.

## 6. Evaluation Methods and Results

Table 3 presents an effect comparison of relevant works based on runtime information categorized into three groups based on method. It can be observed that methods in this category achieve relatively good results, with accuracy exceeding 70% in general. This is mainly because real-time access to runtime information provides more comprehensive data, resulting in higher precision in analysis. However, these methods often come with significant real-time overhead. Therefore, some works also evaluate the cost of these methods, such as the time consumed and the efficiency of parameter collection.

Table 4 presents an effect comparison of relevant works using information retrieval methods based on bug reports. This category of works primarily utilizes text processing and topic modeling techniques to identify similarities between text documents. Therefore, the effectiveness of these methods is highly influenced by the quality of the text. Some works also incorporate the developer's expertise and the bug report handling process as modeling factors to improve the results. Overall, the results of this category of methods do not show a significant improvement or decline compared to the methods based on runtime information.

Table 5 presents an effect comparison of relevant works using machine learning methods based on bug reports. This category of methods primarily relies on NLP techniques to convert bug reports into feature vectors, which are then fed into machine learning models for classification. Generally, the accuracy of these methods is slightly lower that that of the methods based on runtime information and those using IR methods with bug reports. Apart from the precision issues inherent in bug reports themselves, the lower effectiveness can also be attributed to the limitations of the machine learning models used for NLP tasks. However, the advantage of using machine learning methods lies in their efficiency in handling large-scale inputs.

**Table 3.** Effect comparison of works based on runtime information.

| Category | Work | Methods | Runtime Information | Effect | Dataset |
|---|---|---|---|---|---|
| Methods based on comparing stack traces | CrashAutomata | N-gram | Stack traces | F measure: 97% | 5.7 k traces from Mozilla |
| | DURFEX | Variable-length N-gram | Stack traces | 93% and 70% less execution time compared with 1, 2-g | 380 k traces from Firefox and Eclipse |
| | FuRong | Levenshtein distance | Stack trace in Android bug log | 93.4% precision and 87.9% accuracy, on average | 91 bugs from 8 Android applications |
| | S3M | biLSTM encoder | Stack traces | 0.96 and 0.76 RR@10 for JetBrain and Netbeans, respectively | 340 k traces from JetBrains and Netbeans |
| | abaci-finder | kstack2vec, BiLSTM | Stack traces | 0.83 F1 score | 17 k traces from syzbot |
| Methods based on analyzing coverage | CRAXTriage | Coverage comparison | Bug execution path | Not mentioned | 11 programs |
| Methods based on comparing contexts | Fast clustering for UA | Clustering in 2D plane | UAF bug context | 12.2 s clustering time | 1.2 K samples from IE8 |
| | Clustering based on symbolic analysis | symbolic analysis and clustering | Bug execution path | 50% cases allow for finer-grained analysis | 21 programs |
| | Reranking-based deduplication | TF-IDF, Rebucket | races | ∼7 Stack T0% accuracy | 51 k reports from Launchpad and Firefox 48 |
| | REPT | Hardware tracing,reverse debugging, and taint analysis | Program with bugs | 92% accuracy, on average | 14 programs |
| | POMP | Reverse debugging, taint analysis | Program with bugs | More than 93% bug causes identified | 28 prgrams |
| | POMP++ | reverse debugging, taint analysis | Program with bugs | 12% more data flow recovered | 30 programs |
| | IgorFuzz | Graph similarity calculation, spectral clustering | Crash poc | Achieved the highest F score in 90% of cases | Magma and Moonlight benchmark |
| | Triage based on bug signature | PIN, srcML, bear, C-Reduce | Program with bugs | 99.1% precision | Reports from 7 programs |

**Table 4.** Effect comparison of works using the IR method based on bug reports.

| Work | Methods | Effects | Dataset |
| --- | --- | --- | --- |
| Deduplication through local references | Reducing search space based on temporal locality | Up to 53% recall rate | 74 k from FireFox |
| Time-based deduplication | BM25Fext | 45 k from eclipse | 77% RR@20 |
| FactorLCS | Enhancing LCS using size matching within group weight | ≥70% recall rate | 97 k+ from Firefox and 41 k+ from Eclipse |
| Fusion approach For deduplication | MULAN-based multilabel classification model | 72% recall rate and up to 40% performance improvement | 111 k from Firefox |
| Triaging for very large bug repositories | Text cosine similarity, time window, and document factors | ≥95% original recall and low duplicate recall as a filtration aide; ∼70% recall rate as triaging guide | 246 k from Eclipse, Firefox, and Open Office |
| Deduplication using correlations | Stack trace signature, temporal locality, and crash comment textual similarity | 50% and 47% Recall Rate and 55% and 35% precision for the FireFox and Eclipse datasets, respectively | 1 k+ types from FireFox and MSR and 20 k+ from Eclipse |
| LWE | LDA and word embedding | 0.558 RR@20% | 768 k from Mozilla |
| Refined feature-based deduplication | Resolution field extraction | Not mentioned | 10∼22% recall rate improvement and 7∼18% precision improvement |
| Duplication based on multiple factors | Reasonable parameter selection | 80% TP and 0.01%FP for deduplication | 3 M from syzbot |
| Stack trace similarly aggregation | Aggregate computing | 15% RR@1 improvement | 40 k from Netbeans and 210 k from JetBrains |
| CrashSearch | Locality-sensitive hashing | 11% F-score improvement compared with minor hashing | 1 k from eight real-world programs |
| K-detector | AST comparison | 0.986 AUC on SAP HANA | 10 k dump from SAP HANA |
| Reformulating queries | Three different queries | 42 k from 20 open-source projects | 56.6∼78% duplication detection |
| CosTriage | Reduce cost of assigning bugs | 30% cost reduction | 13 k from Apache, 152 k from Eclipse, 5 k from Linux kernel, and 162 k from Mozilla |
| Duplicate based on contextual approach | Multiple context feature comparison | 11.5% accuracy improvement, 41% Kappa improvement, and 16.8% AUC improvement | 37 k from Android, 43 k from Eclipse, 71 k from Mozilla, and 29 k from OpenOffice |
| Triage based on developer analysis | Unigram model and Kullback–Leibler (KL) divergence | ∼75% precision and ∼40% and ∼52% F1 score | 8 k from Eclipse and 10 k from Mozilla |

**Table 4.** *Cont.*

| Work | Methods | Effects | Dataset |
| --- | --- | --- | --- |
| TopicMiner | Multiple-topic model | 68.7% and 90.8% for top-one and top-five precision, respectively | 27 k from GCC, 42 k from OpenOffice, 46 k from Netbeans, 82 k from Eclipse, and 86 k from Mozilla |
| Triage for non-reproducible bug | Time analysis, priority assignment, and NRFixer | ∼70% precision | Mozilla and Eclipse |
| En-LDA | LDA and entropy calculation | 84% RR@5 for JDT and 58% RR@7 for Firefox | 3 k from Mozilla and 2 k from Eclipse |
| Deduplication by continuous querying | Continuous querying | Over 42% duplication prevention | 222.4 k from Android, App Inventor, Bazaar, Cyanogenmod, Eclipse, K9Mail, Mozilla, MyTrack, OpenOffice, Openstack, Osmand, and Tempest |
| Unified triage framework | Information gain, chi-square statistics, TF-IDF, LDA, and SVM | 49.22%, 85.99%, and 74.89% precision for Eclipse, Baiduinput, and Mooctest, respectively | 2 k reports from Eclipse and 0.2 k reports from Baiduinput |
| Triage based on expertise score | Jaccard and cosine similarity | 89.49% accuracy, 89.53 % precision, 89.42% recall rate, and 89.49% F-score | 41 k reports from Mozilla, Eclipse, Netbeans, Firefox, and Freedesktop |
| RSFH | LDA and graph classification | 0.732 accuracy, 0.871 precision, 0.732 recall rate, and 0.796 F score | 135 k from Bugzilla |
| Feature extraction model for triage | TF-IDF and heuristic feature detection | 2% precision, 4.5% recall rate, and 5.9% F-score improvement | Not mentioned |
| Triage based on principal component analysis | Principal component analysis and entropy-based keyword extraction | 90% top-10 team precision and 67% individual precision | 43 k from a private dataset |
| Intuitionistic fuzzy-set-based triaging | Intuitionistic fuzzy sets (IFS) | 15% 0.93, 0.90, and 0.88 precision for Eclipse, Mozilla, and NetBeans, respectively | 32 k from Eclipse, Mozilla, and NetBeans |
| Quality-based classifier | Multiple-feature extraction | 76% precision, 70% recall rate, and 70% F1 score | 5 k from Jira and Bugzilla |
| Intuitionistic fuzzy-set-based triage | LDA and IFSim | 0.894 accuracy, 0.897 precision, 0.893 recall rate, and 0.896 F1 score for Eclipse | Eclipse |
| TM-FBT | Topic modeling and fuzzy logic | 0.903, 0.887, and 0.851 precision for Eclipse, Mozilla, and NetBeans, respectively | Eclipse, Mozilla, and NetBeans |
| CTEDB | Word2Vec, TextRank, SBERT, and DeBERTaV3 | 66 k from eclipse and 230 k from mozilla | Over 98% accuracy, ∼96% precision, 96% recall rate, and 96% F1 score |

**Table 5.** Effect comparison of works using ML methods based on bug reports.

| Work | Methods | Effects | Dataset |
| --- | --- | --- | --- |
| HMM-based deduplication | Hidden Markov models (HMMs) | 76.5% and 73% average accuracy for Firefox and GNOME, respectively | 1 M from Firefox and 753 k from GNOME |
| Soft alignment model for deduplication | Soft-attention alignment and DNN | 5% RR@K improvement | 25 k from Eclipse, 54 k from Mozilla, 11 k from NetBeans, and 15 k from OpenOffice |
| Dual-channel CNN-based deduplication | Word2vec and dual-channel CNN | Over 0.95 accuracy, recall rate, precision, and F1 score | 90 k from OpenOffice, 246 k from Eclipse, and 184 k from Netbeans |
| Domain knowledge-based deduplication | BM25F and multiple ML models | Up to 92% accuracy | 37 k from Android, 42 k from OpenOffice, 72 k from Mozilla, and 29 k from Eclipse |
| Triage in industrial context | SVM and TF-IDF | 53% accuracy, 59% precision, and 47% recall rate | 2 k from Jira and 9 k from Mozilla |
| Deduplication with manifold correlation features | TF-IDF, BM25, and word2Vec | 2.79∼28.97% RR@5 improvement | 6 k from ArgoUML, 9 k from Apache, and 4 k from SVN |
| Deep-learning-based automatic bug triage | Word2Vec and CNN | 82.83% and 35.83% higher performance in top-one and top-three accuracy, respectively | 24 k from four datasets |
| Semisupervised bug triage | Enhanced naive Bayes classifier | 6% accuracy improvement | 20 k from Eclipse |
| DeepTriage-song | BiLSTM and LSTM | 42.96% top-one accuracy | 200 k from Eclipse and 220 k from Mozilla |
| SeqTriagle | Bidirectional RNN and attention model | 5∼20% accuracy improvement | 210 k from Eclipse, 300 k from Mozilla, and 165 k from Gentoo |
| DBR-CNN | Word embedding and CNN | 0.903 F score and 0.919 accuracy | 1.8 k from Hadoop, 12 k from hdfs, 7 k from Mapreduce, and 22 k from Spark |
| TERFUR | NLP model, vector space model, and merging algorithm | 78.15% accuracy, 78.41% recall rate, and 75.82% F1 score | 0.3 k from Justforfun, 0.3 k from SE-1800, 0.4 k from iShopping, 0.2 k from CloudMusic, and 0.4 k from UBook |
| Triage using categorical features | Naive Bayes classifier | 0.633, 0.584, and 0.38 F score for Netbeans, Freedesktop, and Firefox, respectively | Netbeans, Freedesktop, and Firefox |
| DWEN | Word embedding and DNN | Over 0.7 RR@20 | 700 k from Mozilla and 100 k from OpenOffice |
| Multilabel, dual-output DNN for triaging | Mutilabel classifier | 76% accuracy for team assignment and 55% accuracy for individual assignment | 236 k from a private dataset |

**Table 5.** *Cont.*

| Work | Methods | Effects | Dataset |
| --- | --- | --- | --- |
| Triage Using CNN and RF with Boosting | CNN, and boosting-enhanced random forest (BCR) | 96.34% accuracy, and 96.43% F score | Mozilla, Eclipse, JBoss, OpenFOAM, and Firefox |
| itriage | Tossing sequence model and RNN with GRU | 9.39% top-one accuracy improvement | 210 k from Eclipse, 300 k from Mozilla, and 165 k from Gentoo |
| DeepTriage-mani | Bidirectional RNN and softmax classifier | 34∼47% accuracy | 383 k from Chromium, 314 k from Mozilla Core, and 162 k from Mozilla Firefox |
| Triage based on bug cause | Enhanced LDA | 64% F score | 1k+ from Apache, Eclipse, and Mozilla |
| Partially supervised neural network for deduplication and clustering | Word embedding, bidirectional GRU units, topic clustering, and conditional attention-based deduplication | 0.95 and 0.88 F score for Firefox and JDT, respectively | 17 k from SnapS2R, 46 k from Eclipse, and 34 k from FireFox |
| Triage with high confidence | TF-IDF, one-hot encoding, and logistic regression | 89.75% Precision, and 90.17% recall rate | 11 k from Ericsson |
| T-REC | Vector space model, BM25F, and noisy-or classifier | 76.1% CC@5, 83.6% ACC@10, and 89.7% ACC@20 | 9.5 M from Sidia |
| Developer activity-motivated triage | Work2vec and CNN | 0.7489 top-10 accuracy | 39 k from Eclipse, 15 k from Mozilla, and 19 k from Netbeans |
| AI-based document generation model | LDA and backpropagation | Over 84% accuracy | 3 k from Bugzilla and 41 k from MSR |
| Triage for industrial environments | LDA and DNN | 85.1%, 70.1%, and 92.1% RR@5 for JDT, Platform, and Firefox, respectively | 1 k from JDT, 4 k from Platform, and 13 k from Firefox |
| NLP-based triage | Word2vec and LSTM | ∼78% accuracy | Not mentioned |
| Hierarchical attention network for triage | Word2Ve, GloVe, and hierarchical attention network | 50∼65% accuracy | 633 k from Chromium, 1 M from Core, 1 M from Firefox, 187 k from Netbeans, and 318 k from Eclipse |
| Mixed DNN for triage | LSTM and CNN | 80% and 60% top-five accuracy for Eclipse and Mozilla, respectively | 200 k from Eclipse and 220 k from Mozilla |
| BTCSR | TF-IDF, LDA, random walk, and cooperative SkipGram | 51.26% RR@3, 63.25% RR@5, and 74.14% RR@10 | 14 k from Eclipse, 10 k from Mozilla, 11 k from Netbeans, and 2 k from GCC |
| Triage using GCN | TF-IDF and graph convolutional network | 84%, 72.11%, and 66.5% top-10 accuracy for DT, Platform, and Firefox, respectively | 1 k from JDT, 4 k from Platform, and 37 k from Firefox |

**Table 5.** *Cont.*

| Work | Methods | Effects | Dataset |
| --- | --- | --- | --- |
| HINDBR | Low-dimensional space vector conversion | 2 M from nine datasets | 98.83% accuracy and 97.08% F1 score |
| DABT | Bug dependency graph, LDA, TF-IDF, and SVM | 50% bug fix time reduction | 16 k from JDT, 70 k from LibreOffice, and 112 k from Mozilla |
| One-class classification-based triage | One-class SVM | ∼93% average accuracy and ∼53% average recall rate | 4 k from Platform and 20 k form Firefox |
| SusTriage | Multimodal learning | 69% mean average precision improvement and 61% mean reciprocal rank improvement in Eclipse project | 16 k from Eclipse and 15 k from Mozilla |
| Efficient feature extraction model | TF-IDF-based feature extraction | Android, Eclipse, Mozilla, and OpenOffice | 97% accuracy, precision, recall rate, and F1 score |
| Triage in large-scale industrial contexts | TF-IDF and two-level classifier | Human resource reduction | 78 k reports |
| Triage using CNN-LSTM | CNN and LSTM | 52.4% accuracy | 383 k from Chromium, 314 k from Firefox, and 162k from Mozilla core |
| Multi-triage | AST extractor, context augmenter, text encoder, and AST encoder | 57% accuracy for developer triage and 47% accuracy for bug triage | 81.6 k from aspnetcore, azure-powershell, Eclipse, efcore, elasticserach, mixedrealitytoolkit-unity, monogame, nunit, realm-java, Roslyn, and rxjava |
| Triage based on transfer learning | Transfer learning | 75.2%, 82.7%, 78.2%, and 79.3% accuracy for Chrome, Mozilla Core, Firefox, and a private dataset, respectively | 163 k from Chromium, 186 k from Mozilla Core, 138 k from Mozilla Firefox, and 75 k from a private dataset |
| MSDBT | LSTM | 0.5424 RR@3, 0.6375 RR@5, and 0.745 RR@10 | 14 k from Mozilla, 10 k from Eclipse, 11 k from Netbeans, and 2 k from Gcc |
| ST-DGNN | Joint random walk and graph recurrent convolutional neural network | ∼0.7 F1@k | 150 k from Eclipse and 170 k from Mozilla |
| Deepcrash | frame2vec, Bi-LSTM, and Rebucket | 80.72% F score | 10 k from SAP hana and 47 k from Netbeans |
| Triage using transformer | BERT and transfer learning | Over 60% top-10 accuracy | Eclipse, Firefox, and NetBeans |
| DENATURE | TF-IDF, SVM, and logistic regression | 45 k from Eclipse | 88.8% accuracy |
| XAI-based triage | XAI model | 208 K from Eclipse | Not mentioned |
| CombineIRDL | IR + ML | 1 M from Eclipse, Mozilla, and OpenOffice | 7.1∼11.3% precision improvement |

## 7. Findings and Future Direction

### 7.1. Findings from Existing Works

- The currently used BTS use the approaches based on bug reports to implement deduplication and triage, which is mainly determined by the ease of obtaining and transmitting bug reports. The biggest obstacle to using runtime information-based methods lies in the complete acquisition and format conversion of runtime information, which is also a possible research direction in the future. The similarity measurement used by BTS also generally requires more accurate text matching, which also reduces the effectiveness of automatic deduplication and requires more human resources to complete accurate deduplication and triage.
- Stack trace hash has been widely used in many works(∼50%) due to its ready availability and general benefits. It is helpful in identifying root causes and facilitating quick scenario reconstruction and has a certain level of usability. However, its accuracy in determining bug uniqueness is not high. For example, different paths leading to the same crash point may result in splitting of what should be considered the same bug into different ones. Similarly, identical call sequences with different specific values may result in grouping of bugs that should be considered different.
- Relying solely on coverage information is also inaccurate. This is mainly because there may be new execution paths unrelated to triggering the bug, which can lead to different coverage information for the same bug.
- When using runtime information of a program for bug deduplication and triage, false positives may occur because the same bug may exhibit different crash points, error messages, etc.
- In works based on information retrieval, the main sources of information include the bug's basic attributes, crash dumps, stack traces, etc. Among them, stack trace is the most important analysis component, and almost all works(over 90%) refer to it to some extent.
- In works using machine learning methods based on bug reports, the basic approach aligns well with NLP processing approaches. Therefore, most of these works(∼67%) utilize neural network models such as LSTM and CNN. The key input for such works is the textual description information in the bug report. In some works(∼33%), the abilities of developers are also modeled and extracted as features to enhance the model's recognition capability.
- More than 90% of works use open-source databases as test objects. For authors belonging to certain companies, in addition to public, open-source datasets, they also use the company's datasets, such as JetBrains, Ericsson, etc.
- Generally, works based on runtime information tend to have better performance compared than those based on bug reports, but they also come with greater overhead. This is mainly because the accuracy of bug reports cannot be fully guaranteed.

### 7.2. Future Directions

- Stack trace is a primary source of information for works based on information retrieval. However, existing research has shown that this information may not be sufficient to accurately locate bug characteristics. Therefore, future work can consider studying methods to enrich and strengthen stack traces.
- Works based on bug reports heavily rely on bug descriptions, and the accuracy of these descriptions has a significant impact on the results. Currently, most works lack an evaluation analysis of the availability of bug reports. In future work, it would be beneficial to construct models to evaluate the usability of bug reports.
- Current works using DNN models mainly focus on CNN and LSTM. In future work, consideration of the use of updated models such as transformers can be explored to evaluate their effectiveness.
- Exploring ways to improve the efficiency of collecting runtime information and reducing the complexity of processing is a hopeful research point.

## 8. Conclusions

The continuous iteration and maturation of software development techniques have resulted in two consequences. First, both developers and users have demanded increased robustness and stability of software. Secondly, software development cycles have become shorter, making thorough software testing increasingly challenging. Currently, many organizations maintain bug repositories and bug tracking systems to ensure real-time updates of bugs. Each day, a large number of bugs is discovered and sent to the repository, which imposes a heavy workload on bug fixers. Therefore, effective bug deduplication and classification play a crucial role in software development. Numerous studies have been conducted on how to efficiently deduplicate and classify bugs. This paper first introduces the roadmap of related work on deduplication and triage, including recent research trends, mathematical methods, commonly used data sets, and evaluation parameters. Afterward, the specific implementations of deduplication and triage-related works using different technical roadmaps are listed and explained in detail, and the results are quantitatively compared and evaluated. By summarizing various works related to bug deduplication and triage, this paper proposes some key findings and suggests potential future research directions.

**Author Contributions:** Conceptualization, C.Q. and M.Z.; methodology, C.Q.; software, C.Q. and S.L.; validation, C.Q., Y.N. and S.L.; formal analysis, C.Q.; investigation, C.Q.; resources, M.Z.; data curation, Y.N.; writing—original draft preparation, C.Q.; writing—review and editing, M.Z.; visualization, Y.N.; supervision, H.C.; project administration, Y.N.; funding acquisition, H.C. All authors have read and agreed to the published version of the manuscript.

**Funding:** This research received no external funding.

**Institutional Review Board Statement:** Not applicable.

**Informed Consent Statement:** Not applicable.

**Data Availability Statement:** Data were collected from all the sources that are presented in the References section.

**Conflicts of Interest:** The authors declare no conflict of interest.

## References

1. Anvik, J.; Hiew, L.; Murphy, G.C. Coping with an open bug repository. In Proceedings of the 2005 OOPSLA Workshop on Eclipse Technology Exchange, San Diego, CA, USA, 16–17 October 2005; pp. 35–39.
2. Banerjee, S.; Helmick, J.; Syed, Z.; Cukic, B. Eclipse vs. mozilla: A comparison of two large-scale open source problem report repositories. In Proceedings of the 2015 IEEE 16th International Symposium on High Assurance Systems Engineering, Daytona Beach Shores, FL, USA, 8–10 January 2015; pp. 263–270.
3. Banerjee, S.; Cukic, B. On the cost of mining very large open source repositories. In Proceedings of the 2015 IEEE/ACM 1st International Workshop on Big Data Software Engineering, Florence, Italy, 23 May 2015; pp. 37–43.
4. Angell, R.; Oztalay, B.; DeOrio, A. A topological approach to hardware bug triage. In Proceedings of the 2015 16th International Workshop on Microprocessor and SOC Test and Verification (MTV), Austin, TX, USA, 3–4 December 2015; pp. 20–25.
5. Golagha, M.; Lehnhoff, C.; Pretschner, A.; Ilmberger, H. Failure clustering without coverage. In Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis, Beijing, China, 15–19 July 2019; pp. 134–145.
6. Mu, D.; Wu, Y.; Chen, Y.; Lin, Z.; Yu, C.; Xing, X.; Wang, G. An In-depth Analysis of Duplicated Linux Kernel Bug Reports. In Proceedings of the Network and Distributed Systems Security (NDSS) Symposium 2022, San Diego, CA, USA, 24–28 April 2022.
7. Lee, D.G.; Seo, Y.S. Systematic Review of Bug Report Processing Techniques to Improve Software Management Performance. *J. Inf. Process. Syst.* **2019**, *15*, 967–985.
8. Jahanshahi, H.; Cevik, M.; Mousavi, K.; Başar, A. ADPTriage: Approximate Dynamic Programming for Bug Triage. *arXiv* **2022**, arXiv:2211.00872.
9. Wu, H.; Ma, Y.; Xiang, Z.; Yang, C.; He, K. A spatial—Temporal graph neural network framework for automated software bug triaging. *Knowl.-Based Syst.* **2022**, *241*, 108308. [CrossRef]
10. Zhao, Y.; He, T.; Chen, Z. A unified framework for bug report assignment. *Int. J. Softw. Eng. Knowl. Eng.* **2019**, *29*, 607–628. [CrossRef]
11. Lee, D.G.; Seo, Y.S. Improving bug report triage performance using artificial intelligence based document generation model. *Hum.-Centric Comput. Inf. Sci.* **2020**, *10*, 26. [CrossRef]

12. Neysiani, B.S.; Babamir, S.M. Automatic duplicate bug report detection using information retrieval-based versus machine learning-based approaches. In Proceedings of the 2020 6th International Conference on Web Research (ICWR), Tehran, Iran, 22–23 April 2020; pp. 288–293.
13. Uddin, J.; Ghazali, R.; Deris, M.M.; Naseem, R.; Shah, H. A survey on bug prioritization. *Artif. Intell. Rev.* **2017**, *47*, 145–180. [CrossRef]
14. Sawant, V.B.; Alone, N.V. A survey on various techniques for bug triage. *Int. Res. J. Eng. Technol.* **2015**, *2*, 917–920.
15. Neysiani, B.S.; Babamir, S.M. Methods of feature extraction for detecting the duplicate bug reports in software triage systems. In Proceedings of the International Conference on Information Technology, Communications and Telecommunications (IRICT), Tehran, Iran, 1 March 2016; Volume 2016.
16. Yadav, A.; Singh, S.K. Survey based classification of bug triage approaches. *APTIKOM J. Comput. Sci. Inf. Technol.* **2016**, *1*, 1–11. [CrossRef]
17. Chhabra, D.; Malik, M.; Sharma, S. Literature survey on automatic bug triaging using machine learning techniques. In *Proceedings of the AIP Conference Proceedings*; AIP Publishing LLC: Melville, NY, USA, 2022; Volume 2555, p. 020017.
18. Neysiani, B.S.; Babamir, S.M. Duplicate Detection Models for Bug Reports of Software Triage Systems: A Survey. *Curr. Trends Comput. Sci. Appl.* **2019**, *1*, 128–134.
19. Pandey, N.; Sanyal, D.K.; Hudait, A.; Sen, A. Automated classification of software issue reports using machine learning techniques: An empirical study. *Innov. Syst. Softw. Eng.* **2017**, *13*, 279–297. [CrossRef]
20. Goyal, A.; Sardana, N. Machine learning or information retrieval techniques for bug triaging: Which is better? *e-Inform. Softw. Eng. J.* **2017**, *11*, 117–141.
21. Banerjee, S.; Cukic, B.; Adjeroh, D. Automated duplicate bug report classification using subsequence matching. In Proceedings of the 2012 IEEE 14th International Symposium on High-Assurance Systems Engineering, Omaha, NE, USA, 25–27 October 2012; pp. 74–81.
22. Banerjee, S.; Syed, Z.; Helmick, J.; Cukic, B. A fusion approach for classifying duplicate problem reports. In Proceedings of the 2013 IEEE 24th International Symposium on Software Reliability Engineering (ISSRE), Pasadena, CA, USA, 4–7 November 2013; pp. 208–217.
23. Prifti, T.; Banerjee, S.; Cukic, B. Detecting bug duplicate reports through local references. In Proceedings of the 7th International Conference on Predictive Models in Software Engineering, Banff, AB, Canada, 20–21 September 2011; pp. 1–9.
24. Jiang, H.; Chen, X.; He, T.; Chen, Z.; Li, X. Fuzzy clustering of crowdsourced test reports for apps. *ACM Trans. Internet Technol. (TOIT)* **2018**, *18*, 1–28. [CrossRef]
25. Xia, X.; Lo, D.; Ding, Y.; Al-Kofahi, J.M.; Nguyen, T.N.; Wang, X. Improving automated bug triaging with specialized topic model. *IEEE Trans. Softw. Eng.* **2016**, *43*, 272–297. [CrossRef]
26. Panda, R.R.; Nagwani, N.K. Topic modeling and intuitionistic fuzzy set-based approach for efficient software bug triaging. *Knowl. Inf. Syst.* **2022**, *64*, 3081–3111. [CrossRef]
27. Panda, R.R.; Nagwani, N.K. Classification and intuitionistic fuzzy set based software bug triaging techniques. *J. King Saud Univ.-Comput. Inf. Sci.* **2022**, *34*, 6303–6323. [CrossRef]
28. Jang, J.; Yang, G. A Bug Triage Technique Using Developer-Based Feature Selection and CNN-LSTM Algorithm. *Appl. Sci.* **2022**, *12*, 9358. [CrossRef]
29. Choquette-Choo, C.A.; Sheldon, D.; Proppe, J.; Alphonso-Gibbs, J.; Gupta, H. A multi-label, dual-output deep neural network for automated bug triaging. In Proceedings of the 2019 18th IEEE International Conference on Machine Learning and Applications (ICMLA), Boca Raton, FL, USA, 16–19 December 2019; pp. 937–944.
30. Chauhan, R.; Sharma, S.; Goyal, A. DENATURE: Duplicate detection and type identification in open source bug repositories. *Int. J. Syst. Assur. Eng. Manag.* **2023**, *14*, 275–292. [CrossRef]
31. Jiang, Y.; Su, X.; Treude, C.; Shang, C.; Wang, T. Does Deep Learning improve the performance of duplicate bug report detection? An empirical study. *J. Syst. Softw.* **2023**, *198*, 111607. [CrossRef]
32. Dhaliwal, T.; Khomh, F.; Zou, Y. Classifying field crash reports for fixing bugs: A case study of Mozilla Firefox. In Proceedings of the 2011 27th IEEE International Conference on Software Maintenance (ICSM), Williamsburg, VA, USA, 25–30 September 2011; pp. 333–342.
33. Dang, Y.; Wu, R.; Zhang, H.; Zhang, D.; Nobel, P. Rebucket: A method for clustering duplicate crash reports based on call stack similarity. In Proceedings of the 2012 34th International Conference on Software Engineering (ICSE), Zurich, Switzerland, 2–9 June 2012; pp. 1084–1093.
34. Rodrigues, I.M.; Khvorov, A.; Aloise, D.; Vasiliev, R.; Koznov, D.; Fernandes, E.R.; Chernishev, G.; Luciv, D.; Povarov, N. TraceSim: An Alignment Method for Computing Stack Trace Similarity. *Empir. Softw. Eng.* **2022**, *27*, 53. [CrossRef]
35. Shi, H.; Wang, G.; Fu, Y.; Hu, C.; Song, H.; Dong, J.; Tang, K.; Liang, K. Abaci-finder: Linux kernel crash classification through stack trace similarity learning. *J. Parallel Distrib. Comput.* **2022**, *168*, 70–79. [CrossRef]
36. Dunn, T.; Banerjee, N.K.; Banerjee, S. GPU acceleration of document similarity measures for automated bug triaging. In Proceedings of the 2016 IEEE International Symposium on Software Reliability Engineering Workshops (ISSREW), Ottawa, ON, Canada, 23–27 October 2016; pp. 140–145.
37. Wu, R.; Zhang, H.; Cheung, S.C.; Kim, S. Crashlocator: Locating crashing faults based on crash stacks. In Proceedings of the 2014 International Symposium on Software Testing and Analysis, San Jose, CA, USA, 21–25 July 2014; pp. 204–214.

38. Koopaei, N.E.; Hamou-Lhadj, A. CrashAutomata: An approach for the detection of duplicate crash reports based on generalizable automata. In Proceedings of the CASCON, Markham, ON, Canada, 2–4 November 2015; pp. 201–210.
39. Sabor, K.K.; Hamou-Lhadj, A.; Larsson, A. Durfex: A feature extraction technique for efficient detection of duplicate bug reports. In Proceedings of the 2017 IEEE International Conference on Software Quality, Reliability and Security (QRS), Prague, Czech Republic, 25–29 July 2017; pp. 240–250.
40. Tian, Y.; Yu, S.; Fang, C.; Li, P. FuRong: Fusing report of automated Android testing on multi-devices. In Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering: Companion Proceedings, Seoul, Republic of Korea, 27 June–19 July 2020; pp. 49–52.
41. Khvorov, A.; Vasiliev, R.; Chernishev, G.; Rodrigues, I.M.; Koznov, D.; Povarov, N. S3M: Siamese stack (trace) similarity measure. In Proceedings of the 2021 IEEE/ACM 18th International Conference on Mining Software Repositories (MSR), Madrid, Spain, 17–19 May 2021; pp. 266–270.
42. Yeh, C.C.; Lu, H.L.; Lee, Y.H.; Chou, W.S.; Huang, S.K. CRAXTriage: A coverage based triage system. In Proceedings of the 2017 IEEE Conference on Dependable and Secure Computing, Taipei, Taiwan, 7–10 August 2017; pp. 408–415.
43. Liu, Y. RESTCluster: Automated Crash Clustering for RESTful API. In Proceedings of the 37th IEEE/ACM International Conference on Automated Software Engineering, Rochester, MI, USA, 10–14 October 2022; pp. 1–3.
44. Peng, J.; Zhang, M.; Wang, Q. Deduplication and Exploitability Determination of UAF Vulnerability Samples by Fast Clustering. *KSII Trans. Internet Inf. Syst.* **2016**, *10*, 4933–4956.
45. Pham, V.T.; Khurana, S.; Roy, S.; Roychoudhury, A. Bucketing failing tests via symbolic analysis. In Proceedings of the Fundamental Approaches to Software Engineering: 20th International Conference, FASE 2017, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2017, Uppsala, Sweden, 22–29 April 2017; Proceedings 20; Springer: Berlin/Heidelberg, Germany, 2017; pp. 43–59.
46. Moroo, A.; Aizawa, A.; Hamamoto, T. Reranking-based Crash Report Deduplication. In Proceedings of the SEKE, Pittsburgh, PA, USA, 5–7 July 2017; Volume 17, pp. 507–510.
47. Cui, W.; Peinado, M.; Cha, S.K.; Fratantonio, Y.; Kemerlis, V.P. Retracer: Triaging crashes by reverse execution from partial memory dumps. In Proceedings of the 38th International Conference on Software Engineering, Austin, TX, USA, 14–22 May 2016; pp. 820–831.
48. Eom, K.J.; Paik, J.Y.; Mok, S.K.; Jeon, H.G.; Cho, E.S.; Kim, D.W.; Ryu, J. Automated crash filtering for arm binary programs. In Proceedings of the 2015 IEEE 39th Annual Computer Software and Applications Conference, Taichung, Taiwan, 1–5 July 2015; Volume 2, pp. 478–483.
49. Cui, W.; Ge, X.; Kasikci, B.; Niu, B.; Sharma, U.; Wang, R.; Yun, I. {REPT}: Reverse debugging of failures in deployed software. In Proceedings of the 13th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 18), Carlsbad, CA, USA, 8–10 October 2018; pp. 17–32.
50. Xu, J.; Mu, D.; Xing, X.; Liu, P.; Chen, P.; Mao, B. Postmortem Program Analysis with Hardware-Enhanced Post-Crash Artifacts. In Proceedings of the USENIX Security Symposium, Vancouver, BC, Canada, 16–18 August 2017; pp. 17–32.
51. Mu, D.; Du, Y.; Xu, J.; Xu, J.; Xing, X.; Mao, B.; Liu, P. Pomp++: Facilitating postmortem program diagnosis with value-set analysis. *IEEE Trans. Softw. Eng.* **2019**, *47*, 1929–1942. [CrossRef]
52. Jiang, Z.; Jiang, X.; Hazimeh, A.; Tang, C.; Zhang, C.; Payer, M. Igor: Crash Deduplication Through Root-Cause Clustering. In Proceedings of the 2021 ACM SIGSAC Conference on Computer and Communications Security, Virtual Event, Republic of Korea, 15–19 November 2021; pp. 3318–3336.
53. van Tonder, R.; Kotheimer, J.; Le Goues, C. Semantic crash bucketing. In Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering, Montpellier, France, 3–7 September 2018; pp. 612–622.
54. Zhang, X.; Chen, J.; Feng, C.; Li, R.; Diao, W.; Zhang, K.; Lei, J.; Tang, C. DeFault: Mutual information-based crash triage for massive crashes. In Proceedings of the 44th International Conference on Software Engineering, Pittsburgh, PA, USA, 21–29 May 2022; pp. 635–646.
55. Kallingal Joshy, A.; Le, W. FuzzerAid: Grouping Fuzzed Crashes Based On Fault Signatures. In Proceedings of the 37th IEEE/ACM International Conference on Automated Software Engineering, Rochester, MI, USA, 10–14 October 2022; pp. 1–12.
56. Alawneh, A.; Alazzam, I.M.; Shatnawi, K. Locating Source Code Bugs in Software Information Systems Using Information Retrieval Techniques. *Big Data Cogn. Comput.* **2022**, *6*, 156. [CrossRef]
57. Krasniqi, R.; Do, H. Automatically Capturing Quality-Related Concerns in Bug Report Descriptions for Efficient Bug Triaging. In Proceedings of the International Conference on Evaluation and Assessment in Software Engineering 2022, Gothenburg, Sweden, 13–15 June 2022; pp. 10–19.
58. Lee, C.Y.; Hu, D.D.; Feng, Z.Y.; Yang, C.Z. Mining temporal information to improve duplication detection on bug reports. In Proceedings of the 2015 IIAI 4th International Congress on Advanced Applied Informatics, Okayama, Japan, 12–16 July 2015; pp. 551–555.
59. Wang, S.; Khomh, F.; Zou, Y. Improving bug management using correlations in crash reports. *Empir. Softw. Eng.* **2016**, *21*, 337–367. [CrossRef]
60. Rakha, M.S.; Bezemer, C.P.; Hassan, A.E. Revisiting the performance evaluation of automated approaches for the retrieval of duplicate issue reports. *IEEE Trans. Softw. Eng.* **2017**, *44*, 1245–1268. [CrossRef]

61. Banerjee, S.; Syed, Z.; Helmick, J.; Culp, M.; Ryan, K.; Cukic, B. Automated triaging of very large bug repositories. *Inf. Softw. Technol.* **2017**, *89*, 1–13. [CrossRef]
62. Savidov, G.; Fedotov, A. Casr-Cluster: Crash clustering for Linux applications. In Proceedings of the 2021 Ivannikov Ispras Open Conference (ISPRAS), 2–3 December 2021; pp. 47–51.
63. Budhiraja, A.; Reddy, R.; Shrivastava, M. Lwe: Lda refined word embeddings for duplicate bug report detection. In Proceedings of the 40th International Conference on Software Engineering: Companion Proceeedings, Gothenburg, Sweden, 27 May–3 June 2018; pp. 165–166.
64. Chaparro, O.; Florez, J.M.; Singh, U.; Marcus, A. Reformulating queries for duplicate bug report detection. In Proceedings of the 2019 IEEE 26th International Conference on Software Analysis, Evolution and Reengineering (SANER), Hangzhou, China, 24–27 February 2019; pp. 218–229.
65. Karasov, N.; Khvorov, A.; Vasiliev, R.; Golubev, Y.; Bryksin, T. Aggregation of Stack Trace Similarities for Crash Report Deduplication. *arXiv* **2022**, arXiv:2205.00212.
66. James, K.; Du, Y.; Das, S.; Monrose, F. Separating the Wheat from the Chaff: Using Indexing and Sub-Sequence Mining Techniques to Identify Related Crashes During Bug Triage. In Proceedings of the 2022 IEEE 22nd International Conference on Software Quality, Reliability and Security (QRS), Guangzhou, China, 5–9 December 2022; pp. 31–42.
67. Yang, H.; Xu, Y.; Li, Y.; Choi, H.D. K-Detector: Identifying Duplicate Crash Failures in Large-Scale Software Delivery. In Proceedings of the 2020 IEEE International Symposium on Software Reliability Engineering Workshops (ISSREW), Coimbra, Portugal, 12–15 October 2020; pp. 1–6.
68. Park, J.w.; Lee, M.W.; Kim, J.; Hwang, S.w.; Kim, S. Costriage: A cost-aware triage algorithm for bug reporting systems. In Proceedings of the AAAI Conference on Artificial Intelligence, San Francisco, CA, USA, 7–11 August 2011; Volume 25, pp. 139–144.
69. Hindle, A.; Alipour, A.; Stroulia, E. A contextual approach towards more accurate duplicate bug report detection and ranking. *Empir. Softw. Eng.* **2016**, *21*, 368–410. [CrossRef]
70. Badashian, A.S. Realistic bug triaging. In Proceedings of the 38th International Conference on Software Engineering Companion, Austin, TX, USA, 14–22 May 2016; pp. 847–850.
71. Zhang, T.; Yang, G.; Lee, B.; Chan, A.T. Guiding bug triage through developer analysis in bug reports. *Int. J. Softw. Eng. Knowl. Eng.* **2016**, *26*, 405–431. [CrossRef]
72. Goyal, A. Effective Bug Triage for Non-Reproducible Bugs. In Proceedings of the 2017 IEEE/ACM 39th International Conference on Software Engineering Companion (ICSE-C), Buenos Aries, Argentina, 20–28 May 2017; pp. 487–488.
73. Zhang, W.; Cui, Y.; Yoshida, T. En-lda: An novel approach to automatic bug report assignment with entropy optimized latent dirichlet allocation. *Entropy* **2017**, *19*, 173. [CrossRef]
74. Hindle, A.; Onuczko, C. Preventing duplicate bug reports by continuously querying bug reports. *Empir. Softw. Eng.* **2019**, *24*, 902–936. [CrossRef]
75. Yadav, A.; Singh, S.K.; Suri, J.S. Ranking of software developers based on expertise score for bug triaging. *Inf. Softw. Technol.* **2019**, *112*, 1–17. [CrossRef]
76. Alazzam, I.; Aleroud, A.; Al Latifah, Z.; Karabatis, G. Automatic bug triage in software systems using graph neighborhood relations for feature augmentation. *IEEE Trans. Comput. Soc. Syst.* **2020**, *7*, 1288–1303. [CrossRef]
77. Nath, V.; Sheldon, D.; Alphonso-Gibbs, J. Principal Component Analysis and Entropy-based Selection for the Improvement of Bug Triage. In Proceedings of the 2021 20th IEEE International Conference on Machine Learning and Applications (ICMLA), Virtually Online, 13–15 December 2021; pp. 541–546.
78. Panda, R.R.; Nagwani, N.K. An Improved Software Bug Triaging Approach Based on Topic Modeling and Fuzzy Logic. In *Proceedings of the Third Doctoral Symposium on Computational Intelligence: DoSCI 2022*; Springer: Berlin/Heidelberg, Germany, 2022; pp. 337–346.
79. Wu, X.; Shan, W.; Zheng, W.; Chen, Z.; Ren, T.; Sun, X. An Intelligent Duplicate Bug Report Detection Method Based on Technical Term Extraction. In Proceedings of the 2023 IEEE/ACM International Conference on Automation of Software Test (AST), Melbourne, Australia, 15–16 May 2023; pp. 1–12.
80. Ebrahimi, N.; Trabelsi, A.; Islam, M.S.; Hamou-Lhadj, A.; Khanmohammadi, K. An HMM-based approach for automatic detection and classification of duplicate bug reports. *Inf. Softw. Technol.* **2019**, *113*, 98–109. [CrossRef]
81. Rodrigues, I.M.; Aloise, D.; Fernandes, E.R.; Dagenais, M. A soft alignment model for bug deduplication. In Proceedings of the 17th International Conference on Mining Software Repositories, Virtual Online, 29–30 June 2020; pp. 43–53.
82. He, J.; Xu, L.; Yan, M.; Xia, X.; Lei, Y. Duplicate bug report detection using dual-channel convolutional neural networks. In Proceedings of the 28th International Conference on Program Comprehension, Virtual Online, 13–15 July 2020; pp. 117–127.
83. Aggarwal, K.; Timbers, F.; Rutgers, T.; Hindle, A.; Stroulia, E.; Greiner, R. Detecting duplicate bug reports with software engineering domain knowledge. *J. Softw. Evol. Process* **2017**, *29*, e1821. [CrossRef]
84. Dedík, V.; Rossi, B. Automated bug triaging in an industrial context. In Proceedings of the 2016 42th Euromicro Conference on Software Engineering and Advanced Applications (SEAA), Limassol, Cyprus, 31 August–2 September 2016; pp. 363–367.
85. Lin, M.J.; Yang, C.Z.; Lee, C.Y.; Chen, C.C. Enhancements for duplication detection in bug reports with manifold correlation features. *J. Syst. Softw.* **2016**, *121*, 223–233. [CrossRef]

86. Lee, S.R.; Heo, M.J.; Lee, C.G.; Kim, M.; Jeong, G. Applying deep learning based automatic bug triager to industrial projects. In Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering, Paderborn, Germany, 4–8 September 2017; pp. 926–931.
87. Xuan, J.; Jiang, H.; Ren, Z.; Yan, J.; Luo, Z. Automatic bug triage using semi-supervised text classification. *arXiv* **2017**, arXiv:1704.04769.
88. Song, H.-Z.; Ma, Y.-T. DeepTriage:An Automatic Triage Method for Software Bugs Using Deep Learning. *J. Chin. Comput. Syst.* 2019, *40*, 126–132.
89. Chaparro, O. Improving bug reporting, duplicate detection, and localization. In Proceedings of the 2017 IEEE/ACM 39th International Conference on Software Engineering Companion (ICSE-C), Buenos Aires, Argentina, 20–28 May 2017; pp. 421–424.
90. Xi, S.; Yao, Y.; Xiao, X.; Xu, F.; Lu, J. An effective approach for routing the bug reports to the right fixers. In Proceedings of the 10th Asia-Pacific Symposium on Internetware, Beijing, China, 16 September 2018; pp. 1–10.
91. Xie, Q.; Wen, Z.; Zhu, J.; Gao, C.; Zheng, Z. Detecting duplicate bug reports with convolutional neural networks. In Proceedings of the 2018 25th Asia-Pacific Software Engineering Conference (APSEC), Nara, Japan, 4–7 December 2018; pp. 416–425.
92. Alenezi, M.; Banitaan, S.; Zarour, M. Using categorical features in mining bug tracking systems to assign bug reports. *arXiv* **2018**, arXiv:1804.07803.
93. Budhiraja, A.; Dutta, K.; Reddy, R.; Shrivastava, M. DWEN: Deep word embedding network for duplicate bug report detection in software repositories. In Proceedings of the 40th International Conference on Software Engineering: Companion Proceeedings, Gothenburg, Sweden, 27 May–3 June 2018; pp. 193–194.
94. Kukkar, A.; Mohana, R.; Nayyar, A.; Kim, J.; Kang, B.G.; Chilamkurti, N. A novel deep-learning-based bug severity classification technique using convolutional neural networks and random forest with boosting. *Sensors* **2019**, *19*, 2964. [CrossRef]
95. Xi, S.Q.; Yao, Y.; Xiao, X.S.; Xu, F.; Lv, J. Bug triaging based on tossing sequence modeling. *J. Comput. Sci. Technol.* **2019**, *34*, 942–956. [CrossRef]
96. Mani, S.; Sankaran, A.; Aralikatte, R. Deeptriage: Exploring the effectiveness of deep learning for bug triaging. In Proceedings of the ACM India Joint International Conference on Data Science and Management of Data, Kolkata, India, 3–5 January 2019; pp. 171–179.
97. Catolino, G.; Palomba, F.; Zaidman, A.; Ferrucci, F. Not all bugs are the same: Understanding, characterizing, and classifying bug types. *J. Syst. Softw.* **2019**, *152*, 165–181. [CrossRef]
98. Poddar, L.; Neves, L.; Brendel, W.; Marujo, L.; Tulyakov, S.; Karuturi, P. Train one get one free: Partially supervised neural network for bug report duplicate detection and clustering. *arXiv* **2019**, arXiv:1903.12431.
99. Sarkar, A.; Rigby, P.C.; Bartalos, B. Improving bug triaging with high confidence predictions at ericsson. In Proceedings of the 2019 IEEE International Conference on Software Maintenance and Evolution (ICSME), Cleveland, OH, USA, 29 September–4 October 2019; pp. 81–91.
100. Pahins, C.A.D.L.; D'Morison, F.; Rocha, T.M.; Almeida, L.M.; Batista, A.F.; Souza, D.F. T-REC: Towards accurate bug triage for technical groups. In Proceedings of the 2019 18th IEEE International Conference on Machine Learning and Applications (ICMLA), Boca Raton, FL, USA, 16–19 December 2019; pp. 889–895.
101. Guo, S.; Zhang, X.; Yang, X.; Chen, R.; Guo, C.; Li, H.; Li, T. Developer activity motivated bug triaging: Via convolutional neural network. *Neural Process. Lett.* **2020**, *51*, 2589–2606. [CrossRef]
102. Xiao, G.; Du, X.; Sui, Y.; Yue, T. Hindbr: Heterogeneous information network based duplicate bug report prediction. In Proceedings of the 2020 IEEE 31st International Symposium on Software Reliability Engineering (ISSRE), Coimbra, Portugal, 12–15 October 2020; pp. 195–206.
103. Zhang, W. Efficient bug triage for industrial environments. In Proceedings of the 2020 IEEE International Conference on Software Maintenance and Evolution (ICSME), Adelaide, Australia, 28 September–2 October 2020; pp. 727–735.
104. Russo, F.; Raju, R.; Clarke, C.; Yang, N.; Escalona, A.; Tappert, C.C.; Leider, A. *Software Bug Triage Using Machine Learning and Natural Language Processing*; Pace University: New York, NY, USA, 2020.
105. Neysiani, B.S.; Babamir, S.M.; Aritsugi, M. Efficient feature extraction model for validation performance improvement of duplicate bug report detection in software bug triage systems. *Inf. Softw. Technol.* **2020**, *126*, 106344. [CrossRef]
106. He, H.; Yang, S. Automatic Bug Triage Using Hierarchical Attention Networks. In Proceedings of the 2021 IEEE 21st International Conference on Software Quality, Reliability and Security Companion (QRS-C), Hainan Island, China, 6–10 December 2021; pp. 1043–1049.
107. Wang, H.; Li, Q. Effective Bug Triage Based on a Hybrid Neural Network. In Proceedings of the 2021 28th Asia-Pacific Software Engineering Conference (APSEC), Taipei, Taiwan, 6–9 December 2021; pp. 82–91.
108. Yu, X.; Wan, F.; Du, J.; Jiang, F.; Guo, L.; Lin, J. Bug Triage Model Considering Cooperative and Sequential Relationship. In Proceedings of the Wireless Algorithms, Systems, and Applications: 16th International Conference, WASA 2021, Nanjing, China, 25–27 June 2021; Proceedings, Part II 16; Springer: Berlin/Heidelberg, Germany, 2021; pp. 160–172.
109. Zaidi, S.F.A.; Lee, C.G. Learning graph representation of bug reports to triage bugs using graph convolution network. In Proceedings of the 2021 International Conference on Information Networking (ICOIN), Jeju Island, Republic of Korea, 13–16 January 2021; pp. 504–507.
110. Jahanshahi, H.; Chhabra, K.; Cevik, M.; Baþar, A. DABT: A dependency-aware bug triaging method. In *Evaluation and Assessment in Software Engineering, Proceedings of the 25th International Conference on Evaluation and Assessment in Software Engineering, Virtual Online, 21–24 June 2021*; Association for Computing Machinery: New York, NY, USA, 2021; pp. 504–507.

111. Zaidi, S.F.A.; Lee, C.G. One-class classification based bug triage system to assign a newly added developer. In Proceedings of the 2021 International Conference on Information Networking (ICOIN), Jeju Island, Republic of Korea, 13–16 January 2021; pp. 738–741.
112. Zhang, W.; Zhao, J.; Wang, S. SusTriage: Sustainable Bug Triage with Multi-modal Ensemble Learning. In Proceedings of the IEEE/WIC/ACM International Conference on Web Intelligence and Intelligent Agent Technology, Melbourne, VIC, Australia, 14–17 December 2021; pp. 441–448.
113. Aktaş, E.U. Automated Software Issue Triage in Large Scale Industrial Context. Ph.D. Thesis, Sabanci University, Tuzla, Türkiye, 2021.
114. Aung, T.W.W.; Wan, Y.; Huo, H.; Sui, Y. Multi-triage: A multi-task learning framework for bug triage. *J. Syst. Softw.* **2022**, *184*, 111133. [CrossRef]
115. Yu, X.; Wan, F.; Tang, B.; Zhan, D.; Peng, Q.; Yu, M.; Wang, Z.; Cui, S. Deep Bug Triage Model Based on Multi-head Self-attention Mechanism. In Proceedings of the Computer Supported Cooperative Work and Social Computing: 16th CCF Conference, ChineseCSCW 2021, Xiangtan, China, 26–28 November 2021; Revised Selected Papers, Part II; Springer: Berlin/Heidelberg, Germany, 2022; pp. 107–119.
116. Chao, L.; Qiaoluan, X.; Yong, L.; Yang, X.; Hyun-Deok, C. DeepCrash: Deep metric learning for crash bucketing based on stack trace. In Proceedings of the 6th International Workshop on Machine Learning Techniques for Software Quality Evaluation, Singapore, 18 November 2022; pp. 29–34.
117. Zaidi, S.F.A.; Woo, H.; Lee, C.G. Toward an effective bug triage system using transformers to add new developers. *J. Sens.* **2022**, *2022*, 4347004. [CrossRef]
118. Samir, M.; Sherief, N.; Abdelmoez, W. Improving Bug Assignment and Developer Allocation in Software Engineering through Interpretable Machine Learning Models. *Computers* **2023**, *12*, 128. [CrossRef]