Empirical Software Engineering (2025) 30:135
https://doi.org/10.1007/s10664-025-10694-2

KBL: a golden keywords-based query reformulation approach for bug localization

Biyu Cai[1] · **Weiqin Zou**[1] · **Qianshuang Meng**[1] · **Hui Xu**[1] · **Jingxuan Zhang**[1]

Accepted: 23 June 2025

Abstract
Reformulating initial bug reports to obtain better queries for buggy code retrieval is an important research direction in the bug localization area. Existing query reformulation strategies of bug reports are generally unsupervised and may lack localization guidance, which prevents the generation of better queries for bug localization. Towards this, we propose to develop KBL, a golden keywords-based query reformulation approach for bug localization. Specifically, we first leverage the genetic algorithm and keywords refinement heuristic rules to build a golden keywords benchmark targeted at bug localization. Taking this benchmark as bug localization guidance, we create a keywords classifier for bug reports based on three categories of semantic features. The extracted keywords by the classifier for a bug report are taken as the reformulated start point upon which noise removal and shared keyword expansion with historical bug reports are further performed. The final achieved query, as a replacement for the original bug report, is expected to enhance buggy code retrieval performance. Our experiments show that the contributed keywords benchmark is of high quality in locating bugs, establishing a good basis for further query reformulation to improve localization techniques. Through an analysis of different classifier choices, data balancing strategies, and feature importance, we validate the suitability of the configuration settings for our keyword classifier. A testing dataset of 4,484 bug reports from six projects is used to evaluate our KBL. The results show that KBL is found to substantially outperform both the typical (with a relatively 8%-85% higher Acc@10, 9%-93% higher MAP, and 10%-94% higher MRR), and state-of-the-art (with a relatively 21%-45% higher Acc@10, 31%-47% higher MAP and 32%-50% higher MRR) reformulation strategies. Moreover, based on the reformulated queries of our KBL, the performance of seven representative information retrieval-based bug localization techniques also showed recognizable improvements, including relative increases of 8%-36% in Acc@1, 6%-32% in Acc@5, 4%-24% in Acc@10, 4%-21% in Acc@20, 10%-33% in MAP, and 8%-25% in MRR.

Keywords Bug report · Query reformulation · Golden keywords · Bug localization

Communicated by: Bram Adams.

Extended author information available on the last page of the article

Published online: 05 July 2025

1 Introduction

Software users and testers could submit a bug report to bug tracking systems when they encounter software problems. Developers who are assigned to fix the bug first need to reproduce the bug and locate where it appears. To successfully reproduce and locate the bug, the bug report should provide clear and correct information about the bug; the developer should also have a comprehensive understanding of the bug as well as the software project. In other words, bug localization cannot be said to be an easy task for developers during their daily routine, the situation is even more challenging when they have to handle a large number of bug reports. For example, the Mozilla[1] project received as many as more than 10K bug reports in just two months (Luo et al. 2023).

To reduce the burden of developers on locating bugs, various bug localization techniques have been proposed to automatically identify buggy code elements for given bug reports. As a major category of bug localization, information retrieval-based bug localization (IRBL) techniques attract much attention from researchers and practitioners (Rahman and Roy 2018b; Florez et al. 2021; Lukins et al. 2008; Sisman and Kak 2012; Wang and Lo 2014). The basic idea of IRBL techniques is to take bug localization as an information retrieval task where a bug report is a query, the code base is the corpus, and the localization process equals retrieving a list of relevant code files for the bug report query from the code corpus.

As an important input, the quality of a bug report would greatly affect the retrieval performance of IRBL techniques. Previous studies have found that bug reports especially of open-source projects are of different qualities (Rahman and Roy 2018a; Bettenburg et al. 2008). To improve the locating performance, some researchers propose to reformulate the bug report first and then use the generally better query to do bug localization (Rahman and Roy 2018b; Florez et al. 2021; Kim and Lee 2019a; Chaparro et al. 2017). For instance, Rahman and Roy developed a tool named BLIZZARD that classifies bug reports into different types and applies different reformulation strategies to reformulate the bug report (Rahman and Roy 2018b). Meanwhile, Florez et al. focus on information items of a bug report such as observed behavior (OB), expected behavior (EB), steps to reproduce (S2R), based on which performing query reduction and/or expansion strategies to do bug report reformulation (Florez et al. 2021). Sisman and Kak proposed the Spatial Code Proximity model to measure term-term positional proximity. Candidate terms with close proximity are then used to expand the initial query (Sisman and Kak 2013). Haiduc et al. proposed Refoqus, a technique that automatically selects a more suitable reformulation strategy based on the properties of the incoming query, thereby enabling query expansion of reduction (Haiduc et al. 2013), etc.

While the aforementioned reformulation studies have made some progress in facilitating bug localization, there remains substantial room for further performance improvement. Current query reformulation strategies for bug localization primarily focus on utilizing unsupervised graph-based (e.g., TextRank) or frequency-based ranking algorithms. These algorithms are applied to either the entire bug report, parts of it (e.g., stack traces), or pseudo-relevant code files. Pseudo-relevant code files are a set of files initially assumed to be relevant to a bug report based on a retrieval model, even if their actual relevance is unconfirmed. These algorithms weigh terms and select important ones to either expand (completing extra information) or reduce (removing noise) the initial bug report query. Few studies propose using certain items like Observed Behavior plus Title to replace the whole bug report for buggy-code retrieval or adopting explicit relevance feedback to enhance search effectiveness, where developers are required to provide feedback on retrieval results.

[1] https://bugzilla.mozilla.org/home

The potential problems behind these studies are: (1) Pseudo relevant code files can only work if the original bug report is reasonably strong in its power to retrieve at least some of the relevant code files; and explicit relevance feedback would increase the user's burden, be affected by subjective (or wrong) feedback, and generally suffers from scalability issues. (2) Focusing on specific items such as observed behavior or stack traces would lead to missing useful information from other items of a bug report, which makes them not a good enough data basis for keyword extraction. (3) The nature of unsupervised keyword extraction algorithms can only indicate that the selected terms may be real keywords regarding the content of the software artifacts themselves. Still, they may not be the optimal option for the bug localization task. For example, different bug-related tasks, such as priority prediction, bug triage, or localization, are expected to weigh the terms in bug reports with varying emphasis. It is questionable that the terms obtained by applying, e.g., TextRank, to a bug report would have the same effect on all these tasks.

The aforementioned issues lead us to explore an intriguing possibility: could we potentially achieve better localization hints for bug localization if we build a supervised keyword extraction model on a dataset where keywords are labeled based on their buggy-code-retrieval capability? This forms the foundational assumption of the reformulation technique we proposed in this paper, i.e., the KBL — a golden keywords-based query reformulation approach for bug localization. Specifically, we first construct a golden keywords benchmark for bug reports aimed at providing effective localization guidance, by using a genetic algorithm and keywords refinement heuristic rules towards historical bug reports and their fixing data. One main characteristic of the extracted golden keywords lies in their capacity to guide bug localization, as evidenced by their effectiveness in retrieving the correct buggy code. Taking this benchmark as reformulation guidance, we create a supervised keyword classifier based on three categories of semantic features and use the classifier to obtain potential keywords for newly arrived bug reports. The extracted keywords by the classifier for a bug report are taken as the reformulated start point upon which noise removal and keyword expansion with historical bug reports are further performed. The final query achieved could then be fed into an IR search engine (as a substitution for the initial bug report) to locate buggy code files.

We perform our experiments on six open-source projects with 22,747 bug reports in total. The experimental results show that our constructed golden keywords benchmark is of high quality in locating bugs. Through an analysis of different classifier choices, data balancing strategies, and feature importance, we validate the suitability of the configuration settings for our keyword classifier. Compared to traditional reformulation baselines, the KBL shows a relative improvement of 8%-85% in Acc@10, 9%-93% in MAP, and 10%-94% in MRR. In comparison to state-of-the-art reformulation strategies, KBL demonstrates a relative improvement of 21%-45% in Acc@10, 31%-47% in MAP, and 32%-50% in MRR. Moreover, with the leveraging of the reformulated queries generated by our KBL, the performance of seven representative IRBL techniques demonstrated noticeable improvements, with relative increases of 8%-36% in Acc@1, 6%-32% in Acc@5, 4%-24% in Acc@10, 4%-21% in Acc@20, 10%-33% in MAP, and 8%-25% in MRR.

In summary, our study mainly makes the following contributions:

1) We propose to develop KBL, a golden keywords-based reformulation technique for bug localization. By using keywords that provide localization guidance as reformulation start, KBL could generate better bug report queries for locating bugs.

2) We develop an effective way that integrates genetic algorithms and keywords refinement heuristic rules to create a golden keywords benchmark, which provides a valuable starting point for researchers who aim to develop more advanced reformulation techniques tailored for bug localization.
3) We conduct a comprehensive evaluation of KBL based on bug localization tasks, using a testing dataset of 4,484 bug reports. The results demonstrate that KBL outperforms both traditional and state-of-the-art reformulation strategies in key performance metrics; leveraging reformulated queries by KBL enhanced the performance of several representative IRBL techniques.

2 Background and Related Work

In this section, we will first describe the concept of information retrieval-based bug localization and associated research studies. Then, we present current query reformulation strategies and their use in bug localization. Last, we introduce the genetic algorithms that we use to construct golden keywords for reformulation-based bug localization in our study. Details are as follows.

2.1 Information Retrieval-Based Bug localization

Information retrieval bug localization techniques are one main-stream category of existing bug localization techniques. To provide a clearer understanding of the role and importance of IRBL techniques, we begin by briefly introducing the categorization of bug localization methods. Broadly speaking, existing bug localization techniques can be roughly divided into two categories: dynamic bug localization and static bug localization (Le et al. 2015).

Dynamic bug localization mainly leverages program execution traces to associate code elements with program failures (Saha et al. 2013). These techniques typically begin with instrumenting the program to collect execution data during test runs, which contain detailed logs of which parts of the code are executed during successful and/or failed test cases. By comparing the behavior of passing (successful) and failing test cases, a suspiciousness score is computed and assigned to individual code elements (such as lines, methods, etc.). These code elements are then ranked based on their suspiciousness scores, with higher scores indicating larger probabilities of being buggy. Dynamic bug localization's main advantage lies in its ability to leverage real execution data to provide fine-grained fault localization, often down to the line or method level. Accompanying, this precision comes at the cost of higher computational resources and increased execution overhead. Spectrum-based bug localization techniques are representatives of such techniques (Jones and Harrold 2005; Yoo et al. 2017).

Static techniques focus on analyzing static artifacts like source code and bug reports without requiring program execution. For static bug localization, some static semantic features of bug reports and code snippets are extracted first. Then, semantic similarities of those features are calculated and used to locate buggy code elements, where a code element with a higher semantic similarity with a bug report is considered more relevant to the bug (Zhou et al. 2012; Ye et al. 2016). These techniques are helpful for situations where runtime data is unavailable, difficult to collect, or when resource efficiency is paramount. They can quickly scan large codebases and provide a broader overview to help developers identify relevant

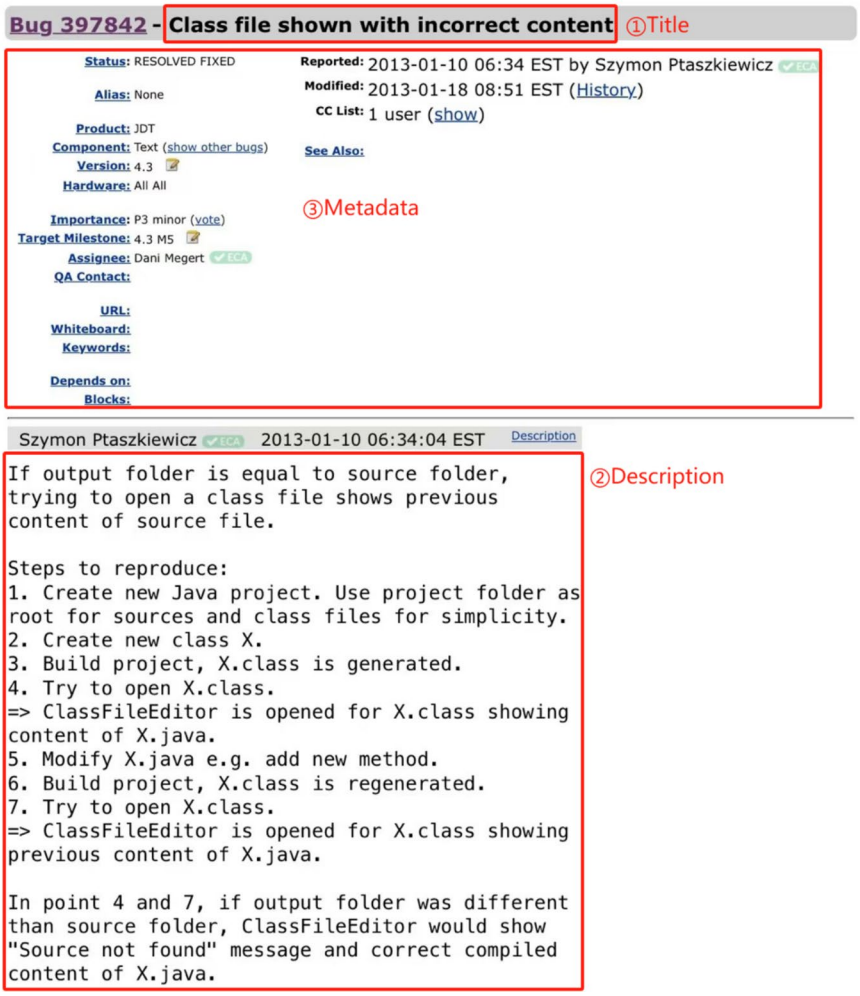


Fig. 1 A bug report example with bugID=397842 in Eclipse JDT project

files or classes that may contain bugs without needing to run the code. Information retrieval-based bug localization (IRBL) techniques are representatives of such techniques (Saha et al. 2013; Kim and Lee 2019a; Chaparro et al. 2017; Zhou et al. 2012; Sisman and Kak 2012).

IRBL treats bug localization as a text retrieval task, where a bug report is treated as a query, code files represent the document collection, and the location process is equivalent to retrieving relevant code documents from the collection for a given bug report. Figure 1 illustrates a real bug report[2] from the Eclipse JDT project. A typical bug report usually consists of (1) bug title, (2) bug description (which may include steps to reproduce and observed behavior), and (3) metadata (containing information such as status, report time, etc.).

For IRBL, bug reports and code files are generally taken as textual contents whose semantics are extracted with traditional information retrieval techniques such as vector space model

[2] https://bugs.eclipse.org/bugs/show_bug.cgi?id=397842

(VSM) (Rao and Kak 2011; Zhou et al. 2012), latent dirichlet allocation (LDA) (Blei et al. 2003; Lukins et al. 2008), and latent semantic indexing (LSI) (Deerwester et al. 1990). Based on e.g., semantic vectors extracted by VSM for a bug report and a code element, a similarity score (generally a cosine similarity) would be calculated. Code elements with the highest scores are returned as buggy candidates to users for further checking. In the early stage, mainly the contents of bug reports and the code themselves are used to do localization. In recent years, researchers have tried to leverage other information sources such as code version control systems or historical bug reports, to improve localizing performance. For instance, Sisman and Kak integrate defect histories and modification histories into their bug localization technique (Sisman and Kak 2012). Zhou et al. propose BugLocator, which utilizes a revised Vector Space Model (rVSM) to rank files based on the textual similarity between the bug report and the source code, and also leverages information from historically similar bug reports to facilitate bug localization (Zhou et al. 20120. Wang and Lo integrate version history, similar bug reports, and structural information to enhance the localization of buggy files (Wang and Lo 2014). Moreno et al. combine textual and structural similarities between code elements in stack traces and source code files for bug localization (Moreno et al. 2014). AmaLgam+ integrates five information sources to locate bugs, including the code version history, similar bug reports, code structures, stack traces, and reporter information (Wang and Lo 2016). With the aim to bridge the possible lexical gap problem within buggy code retrieval process, Ye et al. propose to introduce word embedding technique to better represent the semantics of bug reports and code snippets, so as to further facilitate bug localization (Ye et al. 2016).

Meanwhile, some researchers propose to further combine machine learning techniques to facilitate bug localization. Kim et al. extract features from bug reports and employ Naive Bayes to predict files to be addressed for each bug report (Kim et al. 2013). Ye et al. introduce an adaptive ranking approach that utilizes features extracted from source code files, API descriptions, bug-fixing history, and code change history (Ye et al. 2014). Lam presents an approach named DNNLOC, which combines a deep neural network with an information retrieval technique (i.e., rVSM) and bug-fixing history to recommend potentially buggy source code files for a given bug report (Lam et al. 2017). Yan et al. propose a just-in-time defect localization tool, applying a classifier with 14 change-level features to identify the buggy change lines in committed changes (Yan et al. 2020). We also focus on IRBL techniques. Our goal is to reformulate a bug report by keeping its most informative words and removing its redundant/noisy terms, so as to facilitate bug localization.

2.2 Query Reformulation

In search engines, if an initial query yields unsatisfactory results, users would generally reformulate the query and use the reformulated one to do the search again. The practice of query reformulation is widely applied in various retrieval and location tasks, including bug localization (Sisman and Kak 2013; Chaparro et al. 2017; Rahman and Roy 2018b; Kim and Lee 2019a), concept location (Rahman and Roy 2017; Chaparro and Marcus 2016; Gay et al. 2009) and feature location (Kevic and Fritz 2014), etc. Over time, a list of techniques have been developed to assist users to do query reformulation (Haiduc et al. 2013; Roldan-Vega et al. 2013). These approaches involve either query expansion (Carpineto and Romano 2012), where additional terms are incorporated to broaden the query, or query reduction (Chaparro and Marcus 2016), where terms unlikely to contribute to the inherent meaning of the query are eliminated to diminish noise.

Sisman and Kak were pioneers in introducing query reformulation to the realm of IR-based bug localization (Sisman and Kak 2013). They achieved query reformulation by extracting terms related to the original query from pseudo relevance feedback and using these terms as extensions to the original query. Chaparro et al. enhance the performance of low-quality queries by identifying observed behavior from the bug report as the reformulated query (Chaparro et al. 2017). Considering the quality of bug reports, Rahman and Roy analyzed both structured and unstructured content, and adopted different query expansion or query reduction strategies based on the variations in bug report quality (Rahman and Roy 2018b). Kim and Lee extended bug reports through attachments, and if the quality of the expanded bug reports remained poor, they further reformulated the query by incorporating relevance feedback for additional expansion (Kim and Lee 2019a).

Gay et al. combined IR-based concept location with explicit relevance feedback to enhance the effectiveness of concept location, thereby reducing the burden on developers to reformulate queries (Gay et al. 2009). Haiduc et al. introduce Refoqus, a technique that automatically selects the most appropriate reformulation strategy based on the characteristics of the incoming query, allowing for query expansion or reduction (Haiduc et al. 2013). Chaparro and Marcus demonstrated that removing certain terms from verbose queries can significantly improve the retrieval effectiveness of concept location (Chaparro and Marcus 2016). Rahman and Roy employed term weighting techniques such as TextRank to select the most important terms from the original query and construct a new query to accomplish the concept location task (Rahman and Roy 2017).

Unlike the above query reformulation strategies which mainly rely on applying unsupervised graph-based or frequency-based ranking algorithms to weigh terms within bug reports (or specific items like stack trace) or pseudo/explicit relevant code files for query expansion or reduction. We propose a supervised reformulation strategy driven by golden keywords that can locate bugs with high accuracy.

2.3 Genetic Algorithm

Genetic algorithms (GAs) are search algorithms based on the principles of natural selection and genetics, introduced by J Holland in the 1970's and inspired by the biological evolution of living beings (Sampson 1976). GAs are stochastic global search optimization methods that simulate the replication, crossover, and mutation that occur in natural selection and inheritance. Starting from an initial population, through random selection, crossover, and mutation operations, a group of individuals that are more suitable for the environment is generated. In this way, they continue to reproduce and evolve from generation to generation and finally converge to a group of individuals that are most suitable for the environment. Thus, GAs are widely used to address complex optimization problems in various research fields including Software Engineering (Rahman et al. 2021). Considering that it is verbose to use the whole text (i.e., summary + description) of a bug report as a search query for bug localization, Mills et al. (2020) conduct an empirical study to explore whether a bug report contains enough information for IRBL tasks. With knowing the buggy code files in advance, the authors apply GAs to bug reports and check whether they can find a set of keywords with which a buggy code file can be located accurately. They obtain a yes answer. Their findings inspire our idea of retrieving bug-indicative keywords as guidance for bug report reformulation to improve bug localization.

3 Methodology

In this work, we propose KBL, a golden keywords-based query reformulation approach for bug localization. KBL consists of four modules, which are illustrated in Fig. 2: golden keywords benchmark construction, keywords classifier construction, query reformulation, and evaluation based on bug localization. KBL takes bug reports and source files as input. The benchmark construction module is primarily responsible for building a golden keywords benchmark using the genetic algorithm and the keywords refinement heuristic rules. The keywords classifier construction module focuses on extracting three categories of semantic features from bug reports and source code files, as well as training the keywords classifier. The query reformulation module builds on the initial keywords extracted by the keyword classifier, performing noise removal and shared keyword expansion with historical similar bug reports to obtain the final query. The evaluation module aims to assess the effectiveness of KBL by testing the performance of reformulated queries in bug localization tasks. Before delving into the details of each module, we will first define several terms that will be referenced throughout the description of our KBL, to facilitate a better understanding of its core ideas. The dedicated algorithms used to identify them are presented in Section 3.1.

1) **Preliminary keywords:** A set of terms ultimately output by the genetic algorithm that uses Effectiveness to measure individual fitness during its evolution process. The evolution process terminates when a constructed individual, i.e., a certain set of terms selected from a bug report, makes one buggy code file ranked 1st in the buggy file recommendation list for the bug report, or when the maximum iteration (set as 30,000) is reached.
2) **Noisy keywords:** A subset of preliminary keywords. Each bug report would have a corresponding set of preliminary keywords after the genetic process. For any term from the preliminary keyword set, if its exclusion would not negatively affect bug localization performance, then it will be added to the noisy keyword set for the bug report.
3) **Low-quality keywords:** A subset of noisy keywords. If the exclusion of a certain noisy keyword would not worsen the buggy-code-retrieval performance for any bug report that contains it, then it would be taken as a low-quality keyword, characterized as non-distinctive and uninformative.
4) **Golden keywords:** The terms retained after removing noisy keywords from a bug report's preliminary keyword set are considered the golden keywords for the report. These golden keywords provide valuable guidance for bug localization and will be utilized in the subsequent keyword classifier building.

3.1 Golden Keywords Benchmark Construction

Initially, we gather bug reports from six open-source projects (shared by Ye et al. 2014) and extract their associated buggy code files from git repositories using bug-fixing commits (Dallmeier and Zimmermann 2007). The entire text, including both the summary and description of bug reports, along with the source code, undergoes preprocessing first (details in Section 4.1). Subsequently, the GA takes the preprocessed bug reports and source code as input to generate preliminary keywords for the bug reports (details in Section 3.1.1). After that, we use keywords refinement heuristic rules to filter out unnecessary terms from the preliminary keywords, ultimately producing cleaned keywords, i.e., the golden keywords (details in Section 3.1.2).

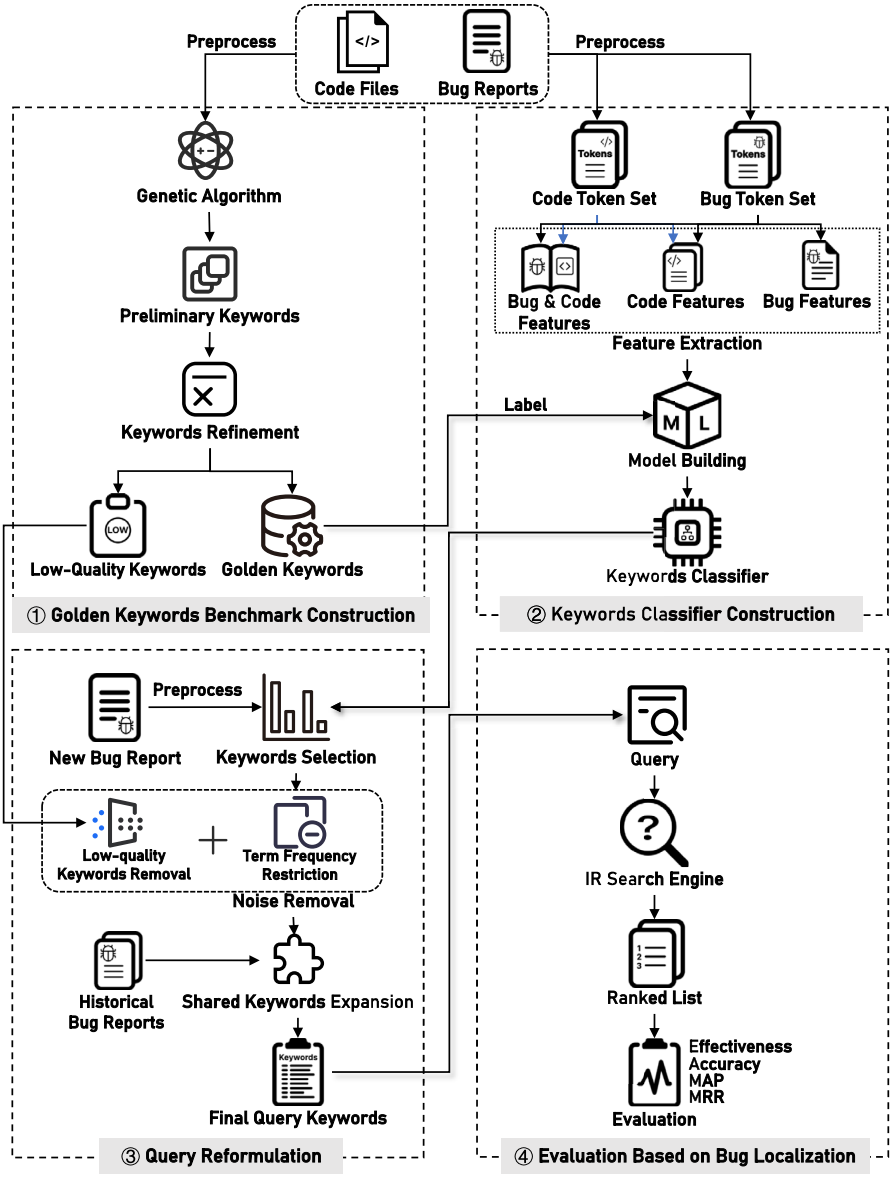


Fig. 2 The overall framework of KBL

3.1.1 Preliminary Keywords Selection

We employ a Genetic Algorithm-based approach to identify high-quality keywords from each bug report for bug localization. As done by an earlier study (Mills et al. 2020), we use a single-objective Genetic Algorithm to generate a high-quality query. The population of individuals is maintained within a search space, where each individual which is coded as a finite length vector represents a solution for the problem. In the context of our problem, each

individual is represented as an integer array, where each element corresponds to a token in the given bug report. If the value of the array element is 1, the corresponding token is part of the query, otherwise, it is 0. The only constraint we set is that the individual must contain at least one "1", that is the formulated query contains at least one term (Mills et al. 2020). To evaluate the individual obtained in every generation, we use the effectiveness metric (the rank of the first truly buggy element in a recommendation list) as the fitness function for our approach. In this way, the GA-based approach will attempt to formulate a query so that the retrieval result can rank the target file at the top. Once the initial generation is created, the algorithm evolves the generation using selection, crossover, and mutation operators as follows.

1) **Selection Operator:** The idea behind selection is to select the individuals with better fitness scores and allow them to pass their genes to successive generations. In our GA-based approach, we choose the roulette selection as the selection operator. Roulette selection is a stochastic method, where the probability for selection of an individual is proportional to its fitness score. That is, the larger the fitness score of an individual is, the more likely it is to be selected.
2) **Crossover Operator:** Crossover means mating between two individuals and reproducing a new individual through choosing crossover sites randomly and exchanging these sites. In our problem context, we perform the selection operator to choose two fittest queries and generate a new query by randomly switching their keywords. In our approach, the one-point crossover strategy is performed. In the one-point crossover, a random crossover point is randomly selected, then all genes (i.e., keywords) behind the point are exchanged between two parent individuals.
3) **Mutation Operator:** Mutation operation is to randomly modify the genes of the newly generated individual. In particular, we randomly choose a gene and modify it by flipping it, which can be translated to removing/adding a term to the query in our problem context. With this operation, the diversity of the population could be increased, and the new population will continue to evolve for the next round.

Our GA-based approach performs the operations above iteratively on each generation until the generated individual achieves the best fitness score (i.e., the effectiveness of the query is 1) or the generation count reaches the preset threshold (i.e., 30,000). After executing the approach, each bug report will obtain a corresponding set of keywords, i.e., the preliminary keywords, as a query, which can achieve near-optimal effectiveness during retrieval (detailed evaluations are presented in Section 5.1).

3.1.2 Keywords Refinement

Although the preliminary keywords have shown good performance in retrieving buggy source code files (i.e., the median values of effectiveness for each project are all 1), we observed that some noise terms still exist within them. To ensure the cleanliness of extracted keyword sets, we propose keywords refinement heuristic rules to remove these redundant and noise terms as much as possible while ensuring that the retrieval performance of keywords is not compromised. Algorithm 1 and its callees Algorithms 2 and 4 shows the details

of how we obtain a clean keyword set from the preliminary keywords. It mainly consists of the following five steps.

Algorithm 1 Keywords refinement.

```
function REFINEMENT(GA_list, BR_list)
    low_quality_keywords ← []
    noisy_list ← []
    GA_list ← []
    for j = 0 to BR_list.length do
        ▷ obtain sub-text set T from a list of sentences S in the given bug report BR_list[j]
        T ← constructSubText(S)
        ▷ obtain sub-keyword set sub_k by labeling T with preliminary keywords in GA and retaining only the terms labeled as key
        sub_k ← constructSubKeywords(T, GA_list[j].keywords)
        ▷ use each element in sub_k as a query to retrieve buggy elements and compute its effectiveness eff, return res[i]=<query, eff>
        res ← retrieveQueries(sub_k)
        ▷ local_low_quality is used to save the local low-quality keywords for the given bug report.
        local_low_quality ← []
        ▷ noisy is used to save the noisy keywords for the given bug report.
        noisy ← []
        ▷ compare the retrieval performance of the two adjacent groups of keywords
        best_k, best_e ← updateBest(res[0].keywords, res[0].eff)
        for i = 1 to res.length − 1 do
            pre ← res[i]
            post ← res[i + 1]
            if pre.query != post.query then
                ▷ update the best keywords
                is_better, is_found, _ ← isBetter(post.query, best_e)
                if is_better then
                    best_k, best_e updateBest(post.keywords, post.eff)
                end if
                ▷ compare two groups of keywords and record the unnecessary tokens
                noisy_res, low_quality_res ← compareKeywords(pre, post)
                noisy.extend(noisy_res)
                local_low_quality.extend(low_quality_res)
            end if
        end for
        low_quality_keywords.extend(local_low_quality)
        noisy_list.append(noisy)
        BR_list[j].best.keywords ← best_k
        BR_list[j].best.eff ← best_e
        BR_list[j].noisy ← noisy
    end for

    low_quality_keywords, updated_noisy ← findLowQualityKeywords(GA_list, low_quality_keywords, noisy_list)
    ▷ Update the noisy keywords using updated_noisy
    BR_list.noisy ← updateNoisy(BR_list, updated_noisy)

    golden_k ← []
    for j = 0 to BR_list do
        best_k ← findBestKeywords(BR_list[j].best, GA_list[j], BR_list[j].noisy)
        golden_k.append(best_k)
    end for
    return golden_k, low_quality_keywords
end function
```

1) **Construct sub-text set:** We treat the text of the summary and the description as the whole text *T* of a given bug report, which contains *n* sentences (i.e., $T = \{s_1, s_2, ..., s_n\}$). To construct the sub-text set, we design a *text window* to select the consecutive sentences as the sub-text. The initial size of the text window is 1, and gradually increases until the end of the text. Specifically, we start from the first sentence of the text *T* and keep expanding the text window until the last sentence is selected, so that we can obtain *n* sub-texts (i.e., $\{s_1, s_1s_2, ..., s_1s_2...s_n\}$). Then, we start from the second sentence, reset the size of the text window to 1 and also keep expanding it, to continue extracting sub-texts (i.e., we would obtain $\{s_2, s_2s_3, ..., s_2s_3...s_n\}$). We repeat the above process until the last sentence of the text *T* is extracted separately as a sub-text. In the end, we can get $\frac{n(n+1)}{2}$ sub-texts to form a sub-text set (Line 7, Algorithm 1).

2) **Construct sub-keywords set and retrieval result:** For each bug report, we have the corresponding preliminary keywords achieved by GA. With the preliminary keywords, we can label every term in the sub-text with *key* or *non-key*. That is, if the term is one of the preliminary keywords, it will be labeled as *key*, otherwise it will be labeled as *non-key*. Then, for each sub-text in the sub-text set, we keep only the terms with *key* label to construct the sub-keywords set (i.e., sub-query set). To find out the noise terms of the preliminary keywords, we still need to perform a retrieval action on each sub-query to obtain the corresponding effectiveness value, which will be used as the basis for comparison between sub-keywords in the following steps(Line 9-11, Algorithm 1).
3) **Compare sub-keywords:** According to our strategy of constructing the sub-text set, there is an inclusion relationship between two adjacent sub-texts in most cases. Correspondingly, there is an inclusion relationship between two adjacent sub-keywords. Therefore, we decide to compare the adjacent sub-keywords of the set in order(Line 17-32, Algorithm 1) and find out the noise tokens based on the token overlap. Specifically, for two sub-keywords k_1, k_2, we first check if their effectiveness value is -1 (Line 2-8, Algorithm 2), if so, the corresponding sub-keywords are regarded as both *noisy keywords* and *local-low-quality keywords* (i.e., terms considered being low quality to the given bug report itself, but not necessarily being low quality for other bug reports that contain them, hence called local). Otherwise, if there exist overlapping keywords of k_1 and k_2, then it will be divided into three situations to handle (Line 15-30, Algorithm 2). The main idea is that the difference between two sub-keywords is the reason for the difference between the effectiveness of the two sub-keywords.

    a) k_1 **is a subset of** k_2**:** If the performance of k_1 is better than k_2, then the complement of k_1 in k_2 are treated as *noisy keywords* (Line 15-18, Algorithm 2).
    b) k_2 **is a subset of** k_1**:** Similarly, if the performance of k_2 is better than k_1, then the complement of k_2 in k_1 is treated as *noisy keywords* (Line 19-22, Algorithm 2).

c) k_1 **intersects** k_2**:** If the performance of k_1 is better than k_2, then the tokens in k_2 but not in k_1 are taken as *noisy keywords*, and the vice versa (Line 23-30, Algorithm 2).

4) **Find out the low-quality keywords:** In the golden keyword benchmark construction process, we also maintain a global set of $low\_quality\_keywords$ for each project, from its historical bug reports. Here, a global low-quality keyword means its exclusion would not worsen the localization performance for any bug report that contains it. This $low\_quality\_keywords$ set could, to some extent, help filter some noise within future bug reports whose golden keywords are not known. Specifically, after step 3), each bug report would have its own set of *local low-quality keywords*, which are then combined into $low\_quality\_keywords$(Line 40, Algorithm 1). To ensure each keyword in $low\_quality\_keywords$ is genuinely low-quality from the global perspective, we will examine the removal of all keywords found in $low\_quality\_keywords$ from the preliminary keywords of each bug report to determine if their removal reduces the bug localization performance (Line 5-22, Algorithm 3). If the performance drops, it suggests that some of the removed keywords (i.e., $removed_k$) might not be truly low-quality. In such cases, these removed keywords are re-added to the preliminary keywords one by one to check if the performance improves; if it does, the re-added keyword is removed from both $low\_quality\_keywords$ and $removed_k$. If not, the keyword is removed again from the preliminary keywords, and the process continues by re-adding the next removed keyword (Line 9-20, Algorithm 3). After validation, the keywords that remain in $removed_k$ are those that can be removed from keywords of the current bug report without causing a decrease in localization performance. The keywords in $removed_k$ are also updated in the noisy keywords of the bug report, since they may originate from the local low-quality keywords of other bug reports (Line 21, Algorithm 3). Upon completing the above verification, the keywords that remain in $low\_quality\_keywords$ are those whose removal from keywords of any bug report does not lead to a decrease in bug localization performance, and thus can be considered true low-quality keywords.
5) **Find out the best keywords:** This step aims to ensure the retrieval effectiveness of the final keywords will not decrease compared to the preliminary keywords. To achieve this, we first compare the effectiveness of the best sub-keywords with the preliminary keywords (Line 6, Algorithm 4) and set the better one as the initial best keywords $best_k$. Next, we remove all *noisy keywords* from $best_k$ and assess if its performance decreases, if not, the filtered keywords are returned as the best keywords (Line 8-11, Algorithm 4). Otherwise, we re-add the non-truly *noisy keywords* back to the keywords and check whether the performance improves (Line 13-23, Algorithm 4). Specifically, we reintroduce keywords from *noisy keywords* one by one, checking the localization performance each time. If re-adding the keyword

improves the performance, the keyword will be retained, and the keywords will be recorded as tmp_best_k. If there is no improvement, the re-added keyword will be removed again and the process continues by re-adding the next removed keyword. Finally, tmp_best_k will be compared with the initial best keywords $best_k$ and the better one will be returned as the best keywords(Line 24-25, Algorithm 4). The returned best keywords are considered as the *golden keywords* for the corresponding bug report. These keywords are expected to locate a bug with good effectiveness performance.

Algorithm 2 Compare two groups of keywords.

Input: k_1, k_2 represents the sub-keywords corresponding to two adjacent sub-texts, respectively.
Output: The identified $noisy\ keywords$ and $low\text{-}quality\ keywords$.

```
1: function COMPAREKEYWORDS(k1, k2)
2:     if k1.eff == −1 and k2.eff == −1 then
3:         return k1.keywords.extend(k2.keywords), k1.keywords.extend(k2.keywords)
4:     else if k1.eff == −1 then
5:         return k1.keywords, k1.keywords
6:     else if k2.eff == −1 then
7:         return k2.keywords, k2.keywords
8:     end if
9:
10:    share_tokens ← k1.keywords ∩ k2.keywords
11:    if share_tokens == null then
12:        return null, null
13:    end if
14:
15:    if k1.keywords.length == share_tokens.length then
16:        ▷ case1: k2 add keywords based on k1
17:        if k1.eff <= k2.eff then noisy ← k2.keywords \ share_tokens
18:        end if
19:    else if k2.keywords.length == share_tokens.length then
20:        ▷ case2: k1 add keywords based on k2
21:        if k1.eff >= k2.eff then noisy ← k1.keywords \ share_tokens
22:        end if
23:    else
24:        ▷ case3: k1 intersects k2
25:        if k1.eff <= k2.eff then
26:            noisy ← k2.keywords \ share_tokens
27:        else
28:            noisy ← k1.keywords \ share_tokens
29:        end if
30:    end if
31:    return noisy, null
32: end function
```

Algorithm 3 Find low-quality keywords.

```
1: function FINDLOWQUALITYKEYWORDS(GA_list, low_quality_set, noisy_list)
2:     ▷ GA_list : the list contain preliminary keywords and effectiveness for each bug report
3:     ▷ low_quality_set : the combined set of local low-quality keywords from all bug reports
4:     ▷ noisy_list : the list containing the noisy keyword sets for each bug report
5:     for i = 0 to GA_list.length do
6:         removed_k ← low_quality_set ∩ GA_list[i].keywords
7:         new_k ← GA_list[i] \ removed_k
8:         is_better, _, new_e ← isBetter(new_k, GA_list[i].eff)
9:         if not is_better then
10:            for j = 0 to removed_k.length do
11:                tmp_k ← new_k.append(removed_k[j])
12:                is_better, _, tmp_e ← isBetter(tmp_k, new_e)
13:                if is_better then
14:                    new_k ← new_k.append(removed_k[j])
15:                    new_e ← tmp_e
16:                    low_quality_set ← low_quality_set.remove(removed_k[j])
17:                    removed_k ← removed_k.remove(removed_k[j])
18:                end if
19:            end for
20:        end if
21:        noisy_list[i] ← noisy_list[i].addAll(removed_k)
22:    end for
23:    return low_quality_set, noisy_list
24: end function
```

Algorithm 4 Find best keywords.

```
1: function FINDBESTKEYWORDS(best, GA, noisy)
2:     ▷ best : the keywords and the corresponding effectiveness with best performance found in the iterations
       for the given bug report
3:     ▷ GA : the keywords and effectiveness achieved by GA algorithm for the given bug report
4:     ▷ noisy : the list of noisy keywords for the given bug report
5:     ▷ initialize the best keywords
6:     best_k, best_e ← comparePerformance(best.keywords, best.eff, GA.keywords, GA.eff)
7:     ▷ remove noisy keywords from best_k
8:     new_k ← best_k \ noisy
9:     is_better, _, new_e ← isBetter(new_k, best_e)
10:    if is_better then
11:        return new_k
12:    else
13:        tmp_best_e ← new_e
14:        for i = 0 to noisy.length do
15:            t ← noisy[i]
16:            is_better, _, new_e ← isBetter(new_k.append(t), tmp_best_e)
17:            if is_better then
18:                tmp_best_k, tmp_best_e ← updateBest(new_k)
19:                noisy ← noisy.remove(t)
20:            else
21:                new_k ← new_k.remove(t)
22:            end if
23:        end for
24:        best_k, best_e ← comparePerformance(tmp_best_k, tmp_best_e, best_k, best_e)
25:        return best_k
26:    end if
27: end function
```

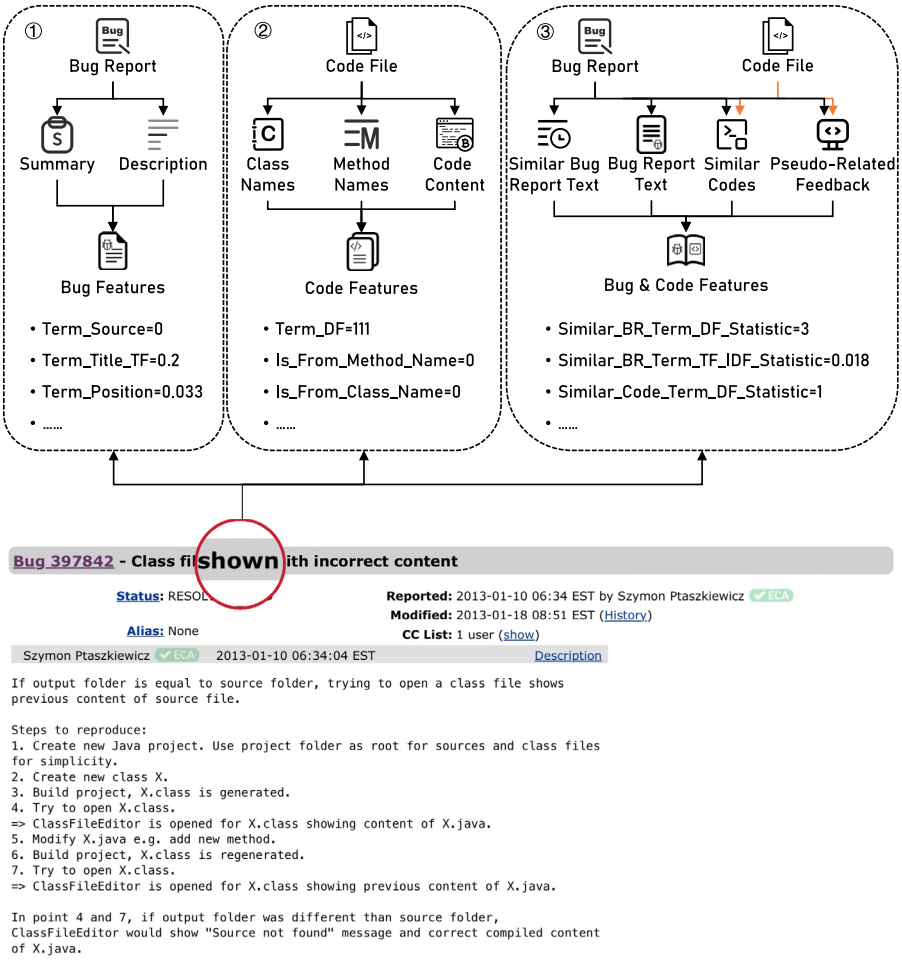


Fig. 3 Differences among Bug Features, Code Features, and Bug&Code Features illustrated at the token level using a bug report (bugID: 397842) on JDT project

After applying the noisy keywords removal, we are expected to get shorter but cleaner or more precise queries that achieve the same or even better performance than the initial queries consisting of the preliminary keywords obtained by GA (This has been validated in our dataset evaluation in Section 5.1).

3.2 Keywords Classifier Construction

This module aims to build a keywords classifier that predicts which tokens of a given bug report are keywords suitable for bug localization. We take keyword classification as a machine learning task, which includes the general three steps, i.e., feature extraction, class labeling and model building. In the feature extraction part, we retrieve three kinds of semantic features for each token of a bug report after preprocessing. Then, we use the

Table 1 Bug Features

Feature Name	Description
Term_Source	Is the term from the title, description, or both of them?
Term_BR_TF	The term frequency of the term in the bug report.
Term_Title_TF	The term frequency of the term appeared in the title.
Term_Span	The distance between the first and the last occurrence of the term in the bug report.
Term_Position	The position information of the term in the bug report.
Part-of-Speech_Tag	The part-of-speech tag of the term.
Term_Meaning_Variety	The number of different meanings that the term has.
Is_From_Camel_Case	Whether the term is obtained by splitting camel case compound terms.
Is_From_Stack_Trace	Whether the term is obtained by splitting camel case compound terms in the stack trace of the bug report.
BR_Term_Co-occurrence	The maximum, mean, and median co-occurrence frequency of a term with other tokens in the bug report.
Term_Dependency_Relationship	The syntactic dependency relationship of the term in the bug report within the corresponding syntactic dependency tree of the sentence it belongs to.
Term_Semantic_Importance	The importance of the term/phrase in the semantic context of the bug report.
Term_Title_Similarity	The similarity between the term/phrase and the title of the bug report.
Similar_BR_Term_Importance	The importance of the term in the top-N historical bug reports similar to the given bug report.

golden keywords benchmark to label each token. After that, we apply typical machine learning algorithms on labeled token instances to build prediction models.

3.2.1 Feature Extraction

We aim to extract the important features for keyword identification from bug reports and source code files. We extracted 61 features and divided them into three categories based on

the type of documents involved in their calculation. That is: (1) `Bug Features`; (2) `Code Features`; (3) `Bug&Code Features`. Specifically, if the information used to calculate a feature solely comes from a bug report, we place the feature into the `Bug Features` category. Similarly, if the calculation of a feature only relies on the content of code files, then we place the feature into the `Code Features` category. For the remaining features whose calculation involves both the contents of bug reports and code files, we place them into the `Bug&Code Features` category. Figure 3 provides an example using the term "shown" of a bug report (with bugID 397842) from the Eclipse JDT project, to illustrate the concrete data sources referred to during the calculation of three feature groups at the token level for each bug report.

(1) Bug Features Table 1 illustrates the features we extracted from bug reports alone, including the feature name and its brief description. More details of each feature are as follows.

1) *Term_Source*: The text of the bug report consists of two parts, namely title and description, where the title is the summary of the description. We extract the source information of the terms based on where they appear in the bug report: the title, the description, or both of them.
2) *Term_BR_TF*: Term frequency (i.e., tf) represents the frequency of occurrence of the term in the document, which can be used to characterize the document. It can be computed as $tf = \frac{d_t}{len(d)}$, where d_t is the number of times that term *t* appears in the document *d* and *len*(*d*) is the total number of terms in the document *d*. To extract this feature for each term, we view the whole text of the bug report as the document *d*.
3) *Term_Title_TF*: Different from the $Token\_BR\_TF$ feature, $Token\_Title\_TF$ treats the title of the bug report as a whole text when calculating the term frequency.
4) *Term_Span*: Term span is a feature commonly used in keyword extraction tasks, thus we also extract this feature for each term in the bug report. The formula $span = \frac{last_t - first_t}{len(d)}$ shows how the term span is calculated, where $first_i$ and $last_i$ are the first and the last occurrences of the term *t* in the bug report text respectively. If the term *t* appears only once in the text, the corresponding term span value is 0.
5) *Term_Position*: Previous research has shown that the candidates' position in the document can be viewed as an effective statistical feature for keyword extraction (Kong et al. 2023). For a term *t* in the bug report, we calculate its position feature as the formula $position = \frac{pos_t}{len(d)}$ where pos_t is the position of the first occurrence of the term *t*.

Table 2 The mapping of term dependency relationships to their feature values

Component	Feature Value	Component	Feature Value
root	1	cc	6
nsubj	2	compound	7
doubj	3	advmod	8
prep	4	det	9
probj	5	amod	10
other	0		

Table 3 Code Features

Feature Name	Description
Term_DF	The ratio of code files where the term appears.
Is_From_Method_Name	Whether the term also occurs in the term set obtained by splitting method names of code files.
Is_From_Class_Name	Whether the term also occurs in the term set obtained by splitting class names of code files.
Term_Code_Similarity	The max, mean, and median value of the similarity between the term/phrase and the source code files.
Term_ClassName_Similarity	The max, mean, and median value of the similarity between the term/phrase and the class names of code files.

6) *Part-of-Speech_Tag*: One of the actionable insights in previous study (Rahman et al. 2021) is that the optimal search keywords are more likely to be *noun*. Thus, we take the part-of-speech (i.e., POS) of the term as one of the features. If a term is a noun, we assign the tag as 1, if it is a verb, the corresponding tag is 2, and if it is an adjective, the corresponding tag is 3. Except for these three POS, the corresponding tags of other POS are all 0.
7) *Term_Meaning_Variety*: For many terms, they have many different meanings. We count the number of different meanings of each term as the value of this feature since it may be helpful for us to classify the terms. In particular, we use WordNet[3], implemented in NLTK, to obtain synsets for the token. Each synset represents a specific meaning associated with the token.
8) *Is_From_Camel_Case*: In code files, a considerable part of the identifiers are camel case compound terms. When such compound terms appear in the bug report text, they can be regarded as localization hints. Hence, they could also be hints for localization keyword extraction. Specifically, if a token belongs to camel case compound terms, the value of the $Is\_From\_Camel\_Case$ feature is 1, otherwise, it is 0.
9) *Is_From_Stack_Trace*: Sometimes, bug reports would include stack trace information that also contains localization hints. Hence, we also pay some attention to the terms within stack traces. The following regular expression is used to extract the stack trace in the bug report. Further, considering that stack traces are generally lengthy and also contain much noise (Rahman and Roy 2018b) related to bug localization, we decided to assign higher feature values to those tokens split from camel case compound terms embedded in stack traces. That is, for these compound terms, we assign the feature values of their split tokens to 1, while the values of other tokens in the stack trace and tokens outside the stack trace are all 0.

```
(.*)?(.+)\.(.+)(\((.+)\.java:\d+\)|\(Unknown␣Source\)| \(Native␣Method\))
```

[3] http://www.nltk.org/howto/wordnet.html

Table 4 Bug&Code Features

Feature Name	Description
Term_TF_IDF	The tf-idf value of the term in the bug report.
Similar_BR_Term_Statistic	The value of tf (maximum, mean, and median), df, and tf-idf of the term in the buggy files corresponding to the top-N historical bug reports similar to the given bug report.
Similar_Code_Term_Statistic	The value of tf (maximum, mean, and median), df, and tf-idf of the Term in the top-N source code files similar to the given bug report.
Term_Feedback_Similarity	The maximum, mean, and median value of the similarity between the term/phrase and the pseudo-related feedback files.
Term_Feedback_ClassName_Similarity	The maximum, mean, and median value of the similarity between the term/phrase and the class name of the pseudo-related feedback files.
Feedback_Term_Statistic	The value of tf (maximum, mean, and median), df, and tf-idf of the Term in the top-N code files of the pseudo-related feedback corresponding to the bug reports.

10) *BR_Term_Co-occurrence*: In natural language processing, co-occurrence measures the frequency of two or more terms appearing together in a text corpus. For bug reports, we build a co-occurrence matrix to quantify how often a term appears adjacent to other terms. We then extract the maximum, average, and median co-occurrence counts for each term, we aim to encapsulate the importance of terms within the bug report's context. This feature is chosen for its ability to highlight the significant associations and dependencies between terms, offering valuable insights into the semantic relationships within bug reports.
11) *Term_Dependency_Relationship*: This feature leverages the dependency relationships of the term within the sentence in bug reports for keyword classification. Analyzing the syntactic connections between each term and other terms provides insights into

the grammatical roles, contributing detailed features that enhance the understanding of keyword context and semantics within the given sentences. In particular, we leverage spaCy[4] to identify the dependency relationship for each term, and we map each type of dependency relationship to a specific value as a feature value. These mapping relationships are presented in Table 2.

12) *Term_Semantic_Importance*: We think that if a crucial term is removed from a piece of text, it may severely disrupt the text's meaning. Thus, we assess term importance based on the extent to which removing a term affects the semantic of the text. The more important terms may be more likely to be keywords. To calculate the feature, we use Word2Vec (Church 2017) to embed the entire bug report, employing max pooling for a representation vector. Comparing the vectors with and without the term, we calculate the Cosine Distance using the formula $cos\_dis = 1 - \frac{\mathrm{A}\cdot\mathrm{B}}{||\mathrm{A}||\cdot||\mathrm{B}||}$. A higher cosine distance indicates lower semantic similarity, signifying greater damage to the text's meaning. Besides removing one term at a time, we also consider removing a phrase at a time to measure the semantic importance of a term. That is, we consider a five-term phrase where the to-be-measured term lies in the central position of the phrase. The similar term importance is calculated after removing the whole phrase from the text.
13) *Term_Title_Similarity*: The title of a bug report is the summary of a bug report. If the semantics of a term are similar to the semantics of the title, then the probability of this term being a keyword in the entire text is higher. The feature $Term\_Title\_Similarity$ is used to measure the similarity between a term and the title. Specifically, we also employ Word2Vec and max pooling to represent the term/phrase and the title as two numeric vectors and then calculate their cosine similarity using the formula $cos\_sim = \frac{\mathrm{A}\cdot\mathrm{B}}{||\mathrm{A}||\cdot||\mathrm{B}||}$.
14) *Similar_BR_Term_Importance*: If a term is the keyword of a bug report, then it is very likely to also be the keyword in those bug reports that are similar to the bug report. Hence, we leverage historical similar bug reports to assist keyword classification for a new bug report. For each term of a bug report, we calculate its *Similar_BR_Term_Importance* as the number of times the term appears as a golden keyword in those historical reports that are similar to the bug report. Given that VSM has been shown to outperform word embedding models like Word2Vec in retrieving similar or duplicate bug reports (Chen et al. 2024), we opted to use the revised VSM model proposed by Zhou et al. (2012) for bug report representation. Cosine similarity is calculated with a threshold of 0.6 to filter similar reports, and this approach is consistently applied throughout the paper for identifying similar bug reports; for all other feature calculation cases that require semantic similarity measurements between, e.g., terms and elements like the bug report title, source code files, pseudo-relevant feedback files, and class names, the Word2Vec built on experimental bug report corpus is used instead in this study."**(2) Code Features** Table 3 shows the features whose calculations only rely on the contents of source code files. The details are as follows.

1) *Term_DF*: The document frequency (df) measures how often a term appears in a collection of documents. It is the ratio of documents containing a particular term to the total number of documents in the collection (i.e, $df_t = \frac{num_d}{len(c)}$). We treat each source code file

[4] https://spacy.io/

in the project as a document, and all the source code files form the whole collection. A higher df of a term indicates a lower likelihood of using the term to differentiate a document.

2) *Is_From_Method_Name*: We first extract all method names for each source code file in the project, and match the method names with the terms in the bug report text. If a compound term matches a specific method name, then we suppose it is the localization hint and assign a feature value of 1 to each simpler term obtained by splitting it. If a term cannot match with any method names, then the feature values for its simpler terms are 0.
3) *Is_From_Class_Name*: Similar to the $Is\_From\_Method\_Name$ feature, we extract all the class names and match them with the compound terms in the bug report. If a term matches any class names, the values for the simpler terms obtained by splitting it are 1, otherwise, they are 0.
4) *Term_Code_Similarity*: If a term/phrase aligns closely with the source code file, then it is more likely to be a keyword in the bug report because this term/phrase may more accurately describe the connotation of the bug. We thus compute the similarity between each term/phrase in the bug report and the source code files. In particular, the cosine similarity is employed to compute the maximum, mean and median similarities between each term/phrase in the bug report and each source file.
5) *Term_ClassName_Similarity*: When analyzing source files, we consider the class names since they often indicate the primary purpose of the code. Therefore, we also assess the similarity between each term/phrase in the bug report and the class name of each source file. Similar to the feature $code\_sim$, the maximum, mean, and median similarities are calculated.**(3) Bug&Code Features** Table 4 present the features whose calculations rely on the contents of both the bug reports and code files. They are detailed as follows.

1) *Term_TF_IDF*: Term frequency-inverse document frequency (tf-idf) is widely used to measure how important a term is within a document (the bug report) relative to a corpus (the whole codebase). Tf-idf is computed using the formula $tf\text{-}idf = tf \times idf$, where tf is the term frequency and $idf = \log \frac{len(c)}{num_d}$, the *len*(*c*) is the number of code files, and the num_d is the number of code files containing the term *t*.

2) *Similar_BR_Term_Statistic*: Two similar bug reports may share similar buggy code files. Hence, for a bug report, if a term appears more frequently in the buggy code files corresponding to its similar bug reports, the term is more likely to be a key term. *Similar_BR_Term_Statistic* is used to capture the occurrence information of terms in the buggy code files corresponding to the similar bug reports for the given bug report. In particular, we calculate df and tf-idf for terms in the current bug report within the buggy code files of selected historical reports. Features also include max term frequency, average term frequency, and median term frequency for each term.
3) *Similar_Code_Term_Statistic*: For a given bug report, its semantics may be similar to the code files that it corresponds to, which means that similar code files can also be used to assist in looking for keywords. If a term appears more frequently in code files similar to a bug report, then the term is more likely to be the keyword of the bug report. Specifically, we first calculate the similarity of the bug report and the source code files using cosine similarity and keep the top-K (i.e., K=10) code files. The df and tf-idf

values, as well as the maximum, mean, and median term frequency for each term are then calculated.

4) *Term_Feedback_Similarity*: The feature $Term\_Code\_Similarity$ considers the similarity between the term and source code files. We also take the similarity between the term and pseudo-relevance feedback (Haiduc et al. 2013) into consideration. Pseudo-relevance feedback represents the initial top-k code files returned by searching the engine with the original query (since no user feedback for result validation, hence called Pseudo-relevance). These code files are considered useful to reformulate a query (Rahman and Roy 2017). If a term of a bug report is semantically similar to the feedback files, the term is likely to be keywords for the bug report. In this study, we query the search engine with the entire bug report text to obtain the top-10 code files as Pseudo-relevance feedback files. Then we calculate the maximum, mean, and median similarities between each term/phrase in the bug report and the 10 feedback files.
5) *Term_Feedback_ClassName_Similarity*: In addition to assessing term similarity with pseudo-relevance feedback files, we also evaluate term similarity with class names in these feedback files. Specifically, we calculate the maximum, mean, and median similarities between each term/phrase in the bug report and the class names of these feedback files.
6) *Feedback_Term_Statistic*: We also suppose the terms frequently appearing in pseudo-relevance feedback files can be treated as potential keywords in the query text. These terms are believed to reflect the topics related to the query. Hence, for each term in a bug report, we introduce the feature $Feedback\_Term\_Statistic$ to capture its occurrence information in the pseudo-relevance feedback files. Specifically, for each term, we would calculate its df and tf-idf value in the top-10 pseudo-relevance feedback files. We also track each term's maximum, mean and median term frequencies in these pseudo-relevance feedback files.

3.2.2 Class Labeling

For a given set of bug reports, we could obtain a list of tokens after preprocessing. As shown in the feature extraction part, 61 features would be calculated for these tokens. The next step is to label these tokens so that they could work as training instances for following model building. In this study, we would label a token as *key* or *non-key*. The label of a token from a bug report is determined by whether the token appears in the golden keywords set of the bug report (which are obtained through the approach described in Section 3.1). In other words, a token is labeled as *key* if it appears in the golden keywords of the bug report it belongs to; otherwise, it is labeled as *non-key*.

3.2.3 Model Building

After the feature extraction and class labeling steps, we could then build our keywords classifier. The input of training includes all the features of the training tokens and their corresponding labels. The output is the trained token-level classifier which could classify tokens of a bug report into *key* or *non-key*.

During model building, we would encounter a problem that the dataset is heavily imbalanced among the instance numbers of two classes. This is because a bug report generally

contain a small fraction of keywords suitable for bug localization, with the majority of terms being non-key. For example, among 449,399 training tokens of the Eclipse_Platform_UI project, only 91,548 (<21%) are keywords. Without handling the imbalanced class problem, the obtained machine learning classifiers would exhibit a bias towards the majority class by for example always assigning the majority class labels to testing instances, so as to achieve a high accuracy on the whole. To avoid the potential negative effects, we apply the widely used random under-sampling (RUS) strategy (Shi et al. 2015; Seiffert et al. 2009) to balance the original training dataset. The basic idea of RUS is to randomly remove an instance from the majority class repeatedly until the instance numbers of different classes are balanced. In our case, we randomly remove non-key terms from the training dataset until the number of key terms and non-key terms is equal.

We then build a keywords classifier by applying the LightGBM[5] (a gradient boosting framework that uses tree based learning algorithms) to the balanced training dataset and subsequently apply the obtained classifier to the testing dataset. This process generates probability values for all terms in bug reports, where terms with higher probabilities are more strongly recommended as golden keywords in a bug report.

3.3 Query Reformulation

This section mainly introduces how to obtain a final query for a new bug report with no ground truth about its golden keywords. That is, for a new bug report, the keywords classifier (built on golden/non-golden keywords datasets of historical bug reports) would be applied first to predict which terms are keywords for the bug report. Then, the initial key terms predicted by the keywords classifier are further reduced through the noise removal process and expanded with shared keywords from historical similar bug reports. The terms kept after the above two steps are constructed as the final query for the bug report to do buggy code retrieval. Details about noise removal and keyword expansion are as follows.

Noise Removal This step involves two main actions to those predicted-to-be keywords by the classifier, including term filtering with $low\_quality$ *keywords* and limiting occurrence frequency. Specifically, considering that the predicted-to-be keywords may also contain noise that negatively affects the localization performance, we propose to use the $low\_quality$ *keywords* (identified through Algorithm 3 in Section 3.1.2) from historical bug reports (that are also used to build the keywords classifier) to filter noise terms. According to the definition of $low\_quality$ *keywords* in Section 3, the terms appearing in $low\_quality$ *keywords* are generally non-distinctive and uninformative, hence, we think the terms that contribute nothing positively to the whole historical bug reports in locating bugs are very likely to be noise for future bug reports, and also should be removed. That is, the predicted-to-be keywords that appear in the $low\_quality$ *keywords* of historical bug reports would be filtered out in later buggy-code-retrieval for the current bug report whose golden keywords are not known.

In addition to filtering out noise terms using $low\_quality$ *keywords*, we also impose restrictions on the frequency of keyword repetition. The same term may appear multiple times in a bug report, but the varying frequency of its occurrence in a query may impact

[5] https://github.com/microsoft/LightGBM

Table 5 Basic Statistics about the Domain and Size Scale for Six Experimental Projects

Project	Domain	LOC	# of Java Files
AspectJ	Aspect-oriented programming	1,515,849	6,879
Birt	Business Intelligence and Reporting	3,720,087	9,697
Platform.UI	Software Development Infrastructure for User Interface	4,195,084	6,243
JDT	Java Development	1,114,440	10,544
SWT	Cross-Platform GUI Library	1,256,595	2,795
Tomcat	Web Application Deployment	822,225	2,042

the final retrieval outcome. Therefore, we impose restrictions on the frequency of a term appearing in the query. That is, if a key term appears more frequently than the specified limit (i.e., 4) in the initial key terms, we retain only the specified number of occurrences and discard the excess. If a key term appears equal to or less than the specified limit, we maintain its original count.

Expand with the Shared Keywords We not only utilize the set of $low\_quality$ *keywords* to reduce the initial key terms, but also expand the initial key terms using *shared keywords* from golden keywords of historical similar bug reports. The underlying assumption is that if two bug reports are similar, then the source code files they require fixing should also be similar. If the given bug report contains terms from buggy files corresponding to historical similar bug reports, then those terms might be among the keywords for the bug report. Therefore, we define a set of *shared keywords*, which are common terms appearing in a given bug report and golden keywords of each historical similar one. Specifically, we measure the similarity between a given bug report and historical bug reports using the rVSM model, retaining only the top-3 historical bug reports with a similarity value greater than the threshold (i.e., 0.6) as the historical similar bug report set for the given bug report. If the number of historical bug reports with similarity values exceeding the threshold is smaller than three, we keep all reports with similarity values greater than the threshold. We compare the terms between the given bug report and golden keywords of each historical similar bug report, and we retain the terms that overlap between them.

3.4 Evaluation Based on Bug Localization

Given that the core of KBL is to construct queries specifically tailored for bug localization tasks, it is a natural choice for us to evaluate KBL by examining how effectively its generated queries perform in bug localization. We mainly conduct two kinds of evaluations to assess KBL's performance. The first one is to compare KBL with existing reformulation techniques, while the second one is to examine whether KBL can enhance the localization performance of representative IRBL methods, in terms of locating bugs.

In the first evaluation, we compare KBL with both traditional and state-of-the-art reformulation strategies. That is, We feed the reformulated queries generated by KBL and the reformulation baselines into the Lucene[6] search engine to retrieve source code files related to the bug reports and then compare their localization performance based on key metrics like

[6] https://lucene.apache.org/

Accuracy@K, MAP, and MRR. Lucene is an open-source full-text search engine widely utilized for information retrieval tasks. It stands out as one of the most frequently utilized search engines in previous IRBL research endeavors (Haiduc et al. 2013; Florez et al. 2021; Moreno et al. 2015; Rahman and Roy 2018b). Lucene integrates Boolean search with a vector space model (VSM)-based search methodology, making it capable of delivering comprehensive search results. In our evaluation, we apply the BM25 similarity model in Lucene to calculate similarity scores. The resulting potentially buggy source code files are returned in a prioritized list, providing developers with suggestions for further examination.

The second evaluation is to explore whether KBL-generated queries can improve the performance of representative IRBL techniques. This could help us further understand the potential of KBL's queries in advancing IRBL research. Seven representative IRBL techniques are chosen for evaluation, including BugLocator (Zhou et al. 2012), BRTracer (Wong et al. 2014), Locus (Wen et al. 2016), BLIA (Youm et al. 2015), BLUiR (Zhou et al. 2012), Amalgam (Wang and Lo 2014), and D&C (Koyuncu et al. 2019). During the evaluation, we replaced the original bug report content required by these IRBL techniques with the reformulated queries of our KBL.

4 Experiment Setup

In this section, we first describe how we construct the datasets for experiments. Then we present the performance metrics for KBL evaluation. Last, we introduce four research questions we aim to answer in this study.

4.1 Experimental Datasets

Before presenting the details of constructing experimental datasets, we first briefly summarize the data flow of our KBL, so that it becomes quite clear what data we need to prepare. That is, we first need to collect a bunch of bug reports and match them to their associated buggy code files. With the matched pairs of bug reports and buggy code files, we can run our GA algorithm and keywords refinement heuristic rules to create the golden keywords benchmark. With the bug reports and their corresponding keywords benchmark, we can build a keywords classifier. The keywords classifier would identify some keywords candidates from a newly arrived bug report. After performing noise removal and shared keywords expansion towards those candidates, a final list of keywords is output. These keywords are expected to be good hints in locating bugs and would work as the final query to retrieve buggy code files for the bug report.

The above dataflow indicates that a dataset of bug reports with buggy code files being known is the starting point of our dataset construction. Related to this, we refer to a dataset kindly shared by Ye et al. (2014). This dataset contains 22,747 bug reports from six software projects and provides associated code files touched to fix these bugs. The six projects are from different domains and are of various size scales (as shown in Table 5). We find that some bugs only have adding code files by analyzing bug fixing code, which means no modifications made to the original code but only adding new code files to fix the bug. In this case, we remove these bugs (372 in total) from the dataset since it is inapplicable to running the localization tool.

Table 6 Experimental Dataset

Project	#All Bug Reports	#Used Bug Reports	#Training Set	#Testing Set
AspectJ	593	576	406	116
Birt	4,178	4,063	2,665	813
Platform.UI	6,495	6,401	4,846	1,281
JDT	6,274	6,235	4,687	1,247
SWT	4,151	4,106	3,020	822
Tomcat	1,056	994	746	205
All	22,747	22,375	16,370	4,484

After obtaining the pairs of bug reports and their associated buggy code files, we need to preprocess them to prepare them for the following golden keywords benchmark construction. We follow the common preprocessing steps adopted by mainstream IRBL research (Huo et al. 2019; Kim et al. 2021; Chaparro et al. 2019) to preprocess bug reports and code files, including tokenizing, stopword removing, identifier splitting, and stemming, etc. Specifically, we retrieve the textual summary and the description items of each fixed bug report. Then we tokenize the text into different terms, among which camel case terms and dotted terms would be split into simpler terms (e.g, ToolBarManagerRenderer, org.eclipse.e4−> Tool Bar Manager Renderer, org eclipse e4). After that, non-alphabetic characters are removed, and terms are converted into their lowercase. We also use the standard English stop word list (Lee et al. 2018) to eliminate the terms that frequently appeared and contributed little to text understanding (e.g., the, an). At last, we apply the Porter stemming tool[7] to transform terms into their root case (e.g., downloaded−>download). Code files undergo the same preprocessing steps as those bug reports.

The bug reports and code files after preprocessing are then fed into our GA and keywords refinement algorithms to obtain the golden keywords benchmark. Our GA is built upon the jmetal framework[8]. By following (Mills et al. 2020), our parameter configurations are as follows: population size: 500, crossover probability: 0.9, mutation probability: 1/n (n is the number of terms of a bug report), and maximum number of generations: 30,000. Considering that keyword extraction with GA is a one-time process that can be completed offline in advance for real applications, and that determining a proper value of maximum chromosome size is inherently challenging (Kim and De Weck 2005), we opt not to impose a limit on the maximum chromosomes size for the GA in experiments.

During benchmark construction, we find that there exists a small number of bug reports (i.e., 1,521) for which no keyword set could help locate a truly buggy file within the returned top-10 file list. To ensure the quality of our golden keywords benchmark and the following keywords classifier construction, we decide to remove these bug reports when training the classifier. For each software project, we chronologically order its bug reports based on their reporting time. The first 80% of bug reports are selected as the training set to build the keywords classifier. The remaining 20% are used to evaluate our KBL. The reasons why we use ordered bug reports like the above are: (1) The computation of some term features (used to build the keyword classifier) relies on information from historical bug reports. (2) It is a recognized practice that future data should not appear in the model-building process (i.e., we can only build models based on existing historical data to make future predictions) (Ye et al.

[7] http://www.nltk.org/api/nltk.stem.html

[8] http://jmetal.sourceforge.net

2014; Lam et al. 2017). For each bug report from the testing dataset, the keywords classifier would predict the terms being golden keywords for bug localization. These terms would undergo noise removal and keywords expansion. The final obtained term set would work as the final query to retrieve buggy code files. The retrieval results are then used to evaluate the performance of our KBL. Table 6 shows the basic statistics of our experimental datasets.

4.2 Performance Metrics

We use the following four metrics to evaluate the bug localization performance of the queries generated by our proposed KBL.

1) **Accuracy@K (Acc@K):** It measures the percentage of bug reports for which at least one buggy code file is correctly recommended to developers in the top-K ranked results.
2) **Effectiveness (E):** E represents the rank of the first truly buggy file in the recommendation list for a bug report. It provides a proxy approximation of how much effort developers would make to find a buggy element. The better the reformulated query is, the smaller the corresponding effectiveness value should be.
3) **Mean Average Precision (MAP):** MAP is commonly used to measure an IR technology based on the mean of average precision (AP) of each query in the query set. A higher MAP value generally indicates a better retrieval performance. It can be calculated as follows:

$$MAP = \frac{\sum_{q \in Q} AP(q)}{|Q|} \tag{1}$$

where Q is the query set (i.e., bug reports in this study), and AP represents the average precision over all buggy files in the result list for a query. The way to compute AP is as follows:

$$AP = \sum_{k=1}^{n} \frac{P_k \cdot buggy_k}{N} \tag{2}$$

where k is the rank, n is the recommendation list size, P_k is the precision at the given rank k (i.e., $P_k = \frac{Number\ of\ relevant\ documents\ in\ top\ k}{k}$). N is the number of truly buggy files within the result list, and $buggy_k$ is a binary value showing whether the k^{th} code file is truly buggy or not.

4) **Mean Reciprocal Rank (MRR):** The reciprocal rank of a query is the multiplicative inverse of the rank of the first correctly identified buggy file. MRR is the reciprocal rank averaged over all queries and it can be calculated as follows:

$$MRR = \frac{1}{|Q|} \sum_{q \in Q} \frac{1}{rank_q} \tag{3}$$

where $rank_q$ is the rank of the first correctly returned buggy file in the result list. Similar to MAP, the bigger the MRR value is, the better a technique is.

4.3 Research Questions

We plan to answer the following questions to understand the performance of our KBL.

RQ1. How Good is our Constructed Keywords Benchmark? A keywords benchmark particularly designed for bug localization plays a fundamental role in our supervised query reformulation approach KBL. Hence, it is quite necessary to check whether the benchmark we constructed is a good data basis for building our keywords classifier, and further research in reformulation techniques designed for bug localization.

RQ2. What Impact Do Keywords Classifier Configurations Have on KBL Performance? The keywords classifier built on the golden keywords benchmark holds a key position in our KBL by predicting possible golden keywords for future bug reports. As a supervised classifier, different configurations (such as the choice of classifiers, the data balancing strategies, and the semantic features) may have different impacts on the performance of the keywords classifier, thereby influencing the final bug localization performance of the reformulated queries by KBL. Answering this RQ could help us better understand our KBL and improve it in the future.

RQ3. How Does KBL Perform Compared to Traditional and Advanced Reformulation Approaches in Bug Localization? This RQ aims to assess how KBL, as a reformulation technique, compares with other traditional and advanced reformulation strategies in terms of bug localization effectiveness. We plan to answer two sub-questions that focus on comparing KBL with both typical and state-of-the-art reformulation strategies separately.

RQ3.1. Does KBL Outperform Typical Reformulation Strategies? To the best of our knowledge, a series of IRBL techniques directly use the title or the description, or both as the proxy of the whole bug report during bug localization. Hence, knowing the performance of our KBL over these strategies is the first step to validate the effectiveness of KBL for bug localization.

RQ3.2. Does KBL Perform Better Than The State-of-the-Art Reformulation Approaches? In typical reformulation strategies (RQ3.1), the content of the title or description would generally not be reformulated. The state-of-the-art reformulation approaches proposed to reformulate the title/description themselves by removing embedded noise or complementing additional information. Whether our KBL could perform better than these state-of-the-art

Table 7 The Effectiveness and Average Numbers of Preliminary Keywords and Golden Keywords

	Preliminary Keywords			Golden Keywords		
Project	Median Eff.	Avg. Eff.	Avg. Num.	Median Eff.	Avg. Eff.	Avg. Num.
AspectJ	1.0	85.1	65.31	1.0	84.1	31.74
Tomcat	1.0	35.8	31.63	1.0	35.8	13.55
SWT	1.0	23.3	39.02	1.0	6.6	17.71
Birt	1.0	206.1	37.53	1.0	43.0	18.67
Platform.UI	1.0	28.2	43.62	1.0	7.3	20.88
JDT	1.0	28.5	49.01	1.0	6.1	25.00

reformulation strategies would provide more convincing evidence on the effectiveness of our KBL in bug localization.

RQ4.Could Queries Generated by KBL Further Enhance the Localization Performance of Representative IRBL Techniques? This RQ aims to explore whether KBL-generated queries can improve the performance of established IRBL techniques in locating bugs. Answering this question will help identify how KBL's query reformulation can complement and enhance existing methods in bug localization, offering valuable insights for the development of future localization technologies.

5 Experimental Results

5.1 RQ1. How Good is our Constructed Keywords Benchmark?

This RQ aims to check the quality of our keywords benchmark constructed by applying genetic algorithms and keywords refinement heuristic rules. Our evaluation would include two parts. In the first part, we would check the overall effectiveness of our benchmark used for bug localization. The effectiveness can help us understand its ability to find the first truly buggy code file. Meanwhile, how many keywords on average would be retrieved so that buggy code files could be correctly located is another aspect we are concerned about. After all, it would be more preferred if we could use fewer but good enough terms to well locate bugs.

Table 7 shows the median/average effectiveness and average number of retrieved keywords in the benchmark. To understand the effect of our GA and keyword refinement algorithms, as shown in the table, we constructed two keyword datasets, i.e., the "Preliminary Keywords" obtained by only running genetic algorithms (GA) to bug reports and code files, and "Golden Keywords" obtained by further running our keywords refinement algorithm after the GA. From Table 7, we can find that for both two keyword datasets, the median effectiveness values for bug reports across six projects are 1 (the first truly buggy code file is ranked at the first rank), which means the keywords obtained by GA and our GA+keywords refinement can correctly locate the bugs. The results from only applying GA validate the findings of Mills et al. (2020) that most bug reports contain sufficient information for bug localization (they also use GA for their exploration). Related to the average effectiveness and average number of keywords, we can find that our proposed keyword refinement heuristic rules could substantially improve the results of the GA. For instance, in SWT, the average

Table 8 The Performance of the Golden Keywords Benchmark in Bug Localization

Project	# BR	Acc@1	Acc@5	Acc@10	Acc@20	MAP	MRR
AspectJ	576	71.00%	84.20%	87.84%	90.79%	50.65%	0.77
Birt	4,063	59.98%	74.57%	79.96%	84.29%	48.43%	0.67
Platform.UI	6,401	78.72%	89.93%	93.12%	95.50%	67.65%	0.84
JDT	6,235	78.86%	91.45%	94.14%	96.37%	65.56%	0.84
SWT	4,106	71.28%	86.87%	91.32%	95.05%	64.79%	0.78
Tomcat	994	75.25%	90.44%	93.46%	96.17%	71.13%	0.82

effectiveness value for the preliminary keyword set is 23.3, while the average effectiveness value for the golden keyword set is 6.6, representing a 72% improvement in average localization performance. Across all six projects, the average length of the golden keyword set decreases by an average of 52% compared to the preliminary keyword set.

> Both GA and GA+keyword refinement approaches could retrieve keywords that perform well in finding the first truly buggy code files. Our proposed keyword refinement algorithm could substantially reduce (with a relative decrease of 52%) the number of needed keywords in locating bugs and improve the overall effectiveness of keywords (with a relative improvement of 51%).

In the second part, we would check the localization performance by using our benchmark (obtained by using GA+keyword refinement algorithms) to locate bugs in terms of Accuracy@K, MAP, and MRR. Table 8 shows the locating results of using our constructed benchmark from six projects. From the table, we can easily find that in six projects, the Acc@20 metric exceeds 90% for five of them. This implies that using the golden keywords benchmark, more than 90% of bug reports in these five projects can locate buggy files within the top-20 retrieval results. Moreover, within these five projects, four of them achieve Acc@10 values exceeding 90%, and even two projects can find over 90% buggy files corresponding to bug reports within the top-5 retrieval results. This indicates that the bug localization performance of the golden keywords benchmark we constructed is excellent. The MAP and MRR values are also found to be high as shown in the table. In other words, these retrieved keywords are truly golden keywords that are of high quality and are good localization hints. This also lays a good data basis for us to train a golden keywords classifier for bug reports to guide bug localization.

> Our constructed keywords benchmark performs well in terms of Accuracy@K, MAP, MRR. On average across all six projects, our constructed keywords benchmark achieves 72.51% on Acc@1, 86.24% on Acc@5, 89.97% on Acc@10, 93.02% on Acc@20, 61.36% on MAP, and 0.79 on MRR. This indicates that the keywords are indeed good localization hints and can be perceived as golden keywords to guide the query reformulation for bug locating.

Table 9 Bug Localization Performance of KBL with Different Classifier Settings

Model	Acc@1	Acc@5	Acc@10	Acc@20
LightGBM	**(1000)22.30%**	**(1974)44.02%**	**(2426)54.10%**	**(2885)64.33%**
RF	(956)21.40%	(1936)43.17%	(2404)53.61%	(2869)63.98%
NB	(879)19.60%	(1877)41.85%	(2355)52.52%	(2812)61.71%
LG	(737)16.43%	(1636)36.48%	(2073)46.23%	(2500)55.75%
DT	(588)13.11%	(1409)31.42%	(1802)40.18%	(2234)49.84%
SVM	(947)21.11%	(1923)42.88%	(2367)52.78%	(2826)63.02%

Table 10 The Classification Performance of Keywords Classifier across Six Projects

Project	F1 score	Precision	Recall
AspectJ	0.45	0.31	0.71
Birt	0.42	0.38	0.47
Eclipse_Platform_UI	0.43	0.34	0.58
JDT	0.45	0.34	0.65
SWT	0.46	0.35	0.67
Tomcat	0.42	0.35	0.53

5.2 RQ2. What Impact Do Keywords Classifier Configurations Have on KBL Performance?

The classifier training phase of KBL includes several key configuration items. To gain deeper insights into how these configurations affect KBL's performance, we plan to focus on exploring the following settings: (1) apply various machine learning (ML) techniques to build the keyword classifier and identify the most appropriate classification algorithm; (2) conducting feature importance analysis on the features used for classifier training to identify the optimal feature combination; (3) experimenting with different data balancing algorithms and sampling ratios to balance the training data and identifying the most suitable sampling algorithm and ratio. By analyzing these training-phase configurations, we can better understand and optimize KBL's classifier.

Keywords Classifiers by Applying Different ML Algorithms To determine the most suitable ML algorithm for KBL, we compare the performance of several widely-used classifiers, including Random Forest (RF), Naive Bayes (NB), Logistic Regression (LG), Decision Tree (DT), Support Vector Machine (SVM), and Light Gradient Boosting Machine (LightGBM), trained on all 61 features. Table 9 shows the comparison results among six classifiers. As shown in Table 9, LightGBM consistently outperforms other classifiers across all Acc@K metrics, demonstrating its superiority in bug localization tasks. For instance, in the Acc@1 metric, KBL correctly localizes 1,000 bugs when utilizing LightGBM, which corresponds to an accuracy of 22.30%. In contrast, when using RF, NB, LG, DT, and SVM classifiers, KBL can only localize 956, 879, 737, 588, and 947 bugs, respectively. In other words, when using RF, NB, LG, DT, and SVM instead of LightGBM, the Acc@1 metric shows a relative performance drop of 4%, 12%, 26%, 41%, and 5%, respectively. Note that, during the above classifier comparison, we employ grid search to fine-tune the hyper-parameters for each classifier like (Koyuncu et al. 2019). The best hyper-parameter configuration for LightGBM is as follows: (1) learning_rate: 0.03; (2) feature_fraction: 0.7; (3) num_leaves: 104; (4) *max_depth*: 10; (5) *min_child_weight*: 0.001; (6) *min_child_samples*: 21; (7) *reg_lambda*: 0.001.

> Using LightGBM as the classifier for KBL yields the best localization performance, when compared to other five typical classifiers, including RF, NB, LG, DT, and SVM. In subsequent experiments, LightGBM is used as the default classifier for KBL .

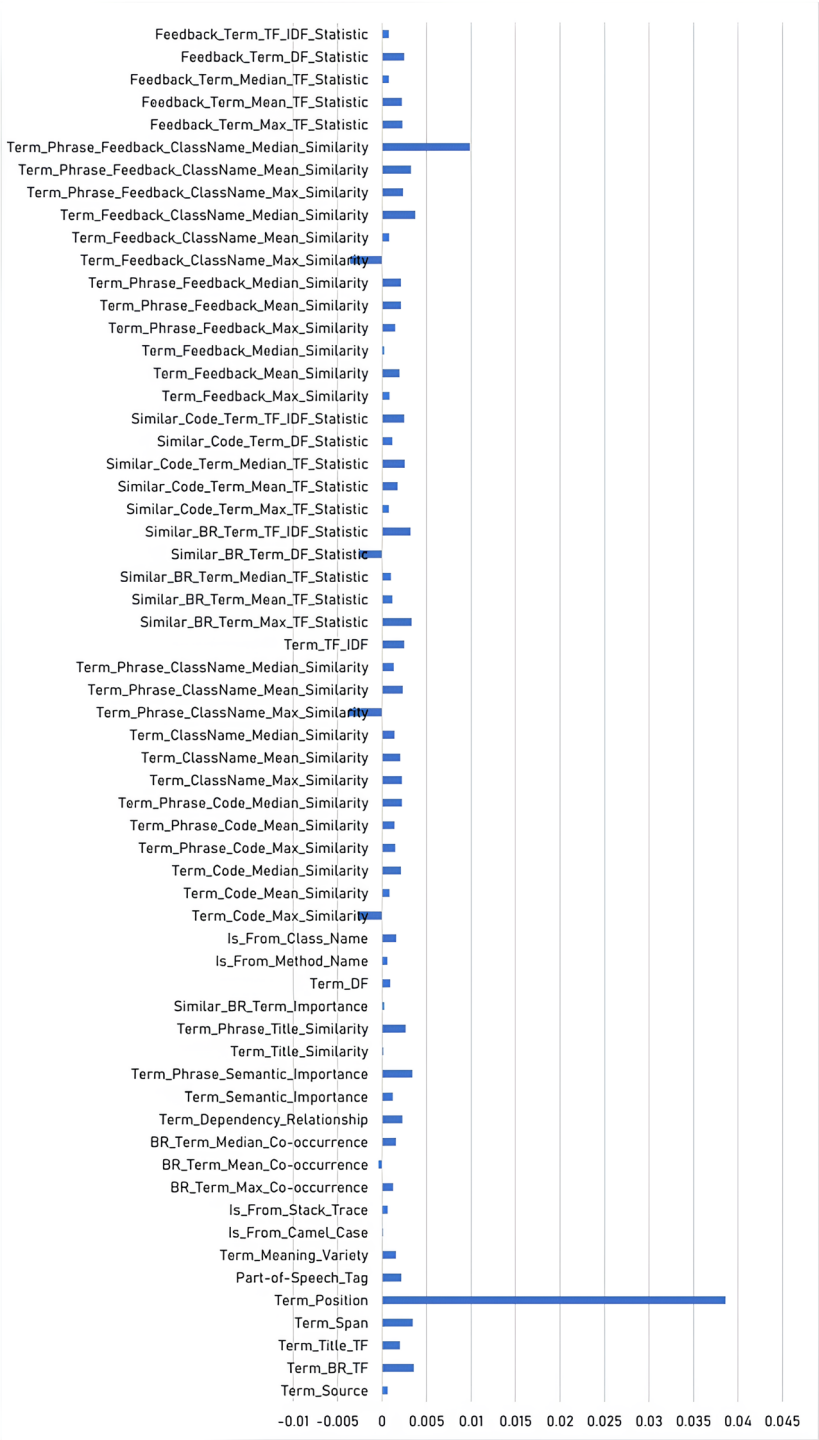


Fig. 4 The impact of 61 features on keyword classifiers in terms of F1-score difference between keyword classifiers with and without each feature in turn

Table 11 The Bug Localization Performance of KBL after Removing Certain Features that Negatively Impact the Keyword Classification Performance

Deleted Feature	Acc@1	Acc@5	Acc@10	Acc@20	MAP	MRR
BR Term Mean Co-occurrence	22.05%	44.42%	53.50%	62.35%	26.45%	0.31
Term Code Max Similarity	21.69%	44.04%	52.09%	61.32%	25.79%	0.31
Term Phrase ClassName Max Similarity	22.19%	44.55%	53.75%	63.44%	26.05%	0.31
Similar BR Term DF Statistic	21.98%	44.09%	53.05%	62.75%	26.27%	0.31
Term Feedback ClassName Max Similarity	22.23%	43.93%	53.99%	62.48%	26.24%	0.31
All Five Features	21.20%	43.01%	52.02%	62.10%	25.81%	0.31
KBL	**22.30%**	**44.02%**	**54.10%**	**64.33%**	**27.03%**	**0.33**

Table 12 Bug Localization Performance of KBL with Keyword Classifier Built on Different Feature Categories

Variant	Acc@1	Acc@5	Acc@10	Acc@20	MAP	MRR
without bug features	20.33%	41.94%	52.34%	62.48%	25.47%	0.31
without code features	19.38%	40.18%	50.31%	60.92%	24.40%	0.30
without bug&code features	20.56%	42.48%	52.78%	62.28%	25.63%	0.31
KBL	**22.30%**	**44.02%**	**54.10%**	**64.33%**	**27.03%**	**0.33**

Table 13 The Number of Keywords and Non-Keywords for Each Project in the Original Training Set

Project	# Non-Keywords	# Keywords
AspectJ	55,531	11,805
Birt	217,747	53,262
Eclipse_Platform_UI	357,851	91,548
JDT	455,163	120,364
SWT	263,422	55,193
Tomcat	44,674	11,034

Feature Importance Analysis To determine the optimal feature combination for KBL, we first conduct a feature importance analysis experiment to identify whether any of the 61 features negatively impact KBL's classification performance. Our feature importance analysis experiment involves constructing new classifiers by iteratively removing individual features to assess their impact on classification performance. Specifically, we compare the F1 score of the new classifier (with one feature removed) to that of the classifier trained with all features (as shown in Table 10, the detail confusion matrices are presented in the Appendix A.1). The difference in F1 scores (i.e., $F1_{all} - F1_{new}$) serves as the importance score for the removed feature. A positive score indicates that classification performance decreased after removing the feature, suggesting that the removed feature positively contributes to the performance of the classifier. Conversely, a negative score suggests that the feature does not contribute positively to the performance of the classifier.

Figure 4 presents the final importance scores of the 61 proposed features, evaluated across six projects (the F1 score difference in the figure is the average difference across six projects). As shown, five features (namely *Term Feedback ClassName Max Similarity*, *Similar BR Term DF Statistic*, *Term Phrase ClassName Max Similarity*, *Term Code Max Similarity*, and *BR Term Mean Co-occurrence*) have negative importance scores, indicating that they

Table 14 Performance Comparison of KBL with Different Data Balancing Techniques

Balancing Technique	Acc@1	Acc@5	Acc@10	Acc@20	MAP	MRR
RUS	**22.30%**	**44.02%**	**54.10%**	**64.33%**	**27.03%**	**0.33**
NCR	19.06%	41.05%	50.75%	60.32%	24.69%	0.30
Tomek Links	16.30%	36.41%	45.98%	55.90%	21.64%	0.26
ROS	17.48%	38.67%	48.81%	58.11%	22.96%	0.28

Table 15 Performance Comparison of KBL when RUS with Different Sampling ratios

Ratio	Acc@1	Acc@5	Acc@10	Acc@20	MAP	MRR
1:1	**22.30%**	**44.02%**	**54.10%**	**64.33%**	**27.03%**	**0.33**
2:1	20.85%	41.07%	51.00%	59.00%	24.06%	0.29
3:1	15.20%	34.99%	45.00%	52.00%	20.74%	0.25
4:1	13.02%	30.03%	41.00%	48.02%	19.32%	0.22
No Balancing	13.00%	30.01%	41.99%	48.03%	19.62%	0.22

negatively impact the performance of the classifier. To further verify the impact of these five features on KBL's bug localization performance, we remove these features individually and collectively to observe any changes in localization performance. Table 11 compares KBL's localization performance with all 61 features against its performance after removing specific features. From the table, we observe that KBL achieves the best bug localization performance when all 61 features are retained. Removing any of these five features, or all of them together, leads to a decline in the localization performance of KBL. For instance, when we individually remove the features *BR Term Mean Co-occurrence*, *Term Code Max Similarity*, *Term Phrase ClassName Max Similarity*, *Similar BR Term DF Statistic*, and *Term Feedback ClassName Max Similarity*, the resulting models show relative declines of 3%, 5%, 1%, 2%, and 3%, respectively, in the Acc@20 metric compared to KBL using all 61 features. When we remove all five features together, the model also show a relative decline of 3% in Acc@20 metric compared to KBL with all 61 features.

Additionally, we further examined the influence of three feature categories (i.e., bug features, code features, bug&code features) on KBL through ablation experiments. Our ablation experiments perform by removing one category of features at a time to build the keyword classifier. Table 12 shows the localization performance of KBL with keyword classifiers built on different feature categories. We can observe that the performance of all three KBL variants is inferior to the KBL using all features. Specifically, compared to variant *without bug features*, KBL relatively improves by 10%, 6%, and 6% in acc@1, MAP, and MRR, respectively. In comparison to the variant *without code features*, KBL relatively improves by 15%, 11%, and 10% in these three metrics, respectively. Compared to variant *without bug&code features*, KBL improves by 8%, 5%, and 6% in these three metrics. We can observe that the removal of code features has a greater impact of KBL compared to the other two variants, while the contribution of bug features and bug&code features to KBL are relatively close.

> KBL achieves the best localization performance when using all 61 features, with the *code features* category contributing more to the localization performance than the other two feature categories.

Data Balancing Using Various Balancing Techniques and Sampling Ratios Balancing the training data is an essential step in the classifier training phase to ensure fair and effective learning. Table 13 shows the distribution of tokens labeled as *key* and *non-key* in the training sets of six projects, revealing a noticeable imbalance in the number of tokens between the two categories. This imbalance would introduce biases in the classifier, affecting its ability to learn effectively from the training data and ultimately impacting KBL's performance in the bug localization task. To mitigate this issue and improve the localization performance of KBL, we first explore the impact of various data balancing techniques on the localization effectiveness of KBL. We experiment with four commonly used data balancing techniques, namely Random Under Sampling (RUS), Tomek Links (Tomek 1976), Neighborhood Cleaning Rule (NCR) (Laurikkala 2001), and Random Over Sampling (ROS), to balance our original training data. RUS involves randomly reducing the number of instances in the majority class to balance the dataset. Tomek Links (Tomek 1976) is a method that removes overlapping instances between classes, refining the dataset by reducing the presence of noisy data in the majority class. NCR (Laurikkala 2001) cleans the dataset by evaluating the neighborhood of each instance to identify and remove noisy instances from the majority

Table 16 Performance of KBL in Bug Localization

Project	Queries	Acc@1	Acc@5	Acc@10	Acc@20	MAP	MRR
AspectJ	Title	11.20%	21.55%	31.89%	38.79%	14.04%	0.17
	Description	12.93%	24.14%	29.31%	37.06%	15.81%	0.19
	Title + Description	18.10%	30.17%	38.79%	50.00%	21.44%	0.26
	KBL	**25.86%**	**37.06%**	**49.13%**	**56.89%**	**27.04%**	**0.32**
Birt	Title	8.61%	19.92%	27.55%	36.77%	11.21%	0.15
	Description	5.16%	10.94%	14.14%	19.06%	6.27%	0.08
	Title + Description	10.33%	21.52%	29.39%	39.36%	12.70%	0.17
	KBL	**12.30%**	**25.58%**	**33.70%**	**42.92%**	**14.66%**	**0.19**
Platform.UI	Title	17.25%	39.73%	49.64%	58.31%	23.50%	0.28
	Description	10.38%	24.12%	29.97%	37.78%	14.29%	0.17
	Title + Description	19.12%	40.74%	49.57%	60.18%	24.55%	0.29
	KBL	**20.92%**	**42.85%**	**53.00%**	**62.99%**	**26.56%**	**0.32**
JDT	Title	20.93%	46.27%	58.05%	70.40%	27.42%	0.33
	Description	12.99%	28.62%	35.20%	40.89%	16.84%	0.20
	Title + Description	23.57%	48.83%	59.58%	69.60%	29.20%	0.35
	KBL	**25.66%**	**52.12%**	**63.03%**	**73.45%**	**31.21%**	**0.38**
SWT	Title	17.63%	37.83%	47.93%	61.67%	24.16%	0.28
	Description	12.16%	26.64%	33.81%	42.33%	16.48%	0.19
	Title + Description	22.62%	43.18%	54.25%	66.05%	28.35%	0.33
	KBL	**24.69%**	**48.41%**	**59.24%**	**71.16%**	**30.09%**	**0.36**
Tomcat	Title	26.82%	52.19%	64.39%	76.09%	33.75%	0.39
	Description	11.70%	25.36%	28.78%	32.68%	15.22%	0.17
	Title + Description	27.80%	57.07%	68.29%	77.07%	35.73%	0.41
	KBL	**38.53%**	**61.46%**	**69.75%**	**79.02%**	**41.34%**	**0.48**
All	Title	17.06%	37.71%	47.88%	58.69%	22.71%	0.27
	Description	10.61%	23.50%	29.19%	35.83%	14.03%	0.17
	Title + Description	19.78%	40.43%	50.13%	60.61%	24.82%	0.30
	KBL	**22.30%**	**44.02%**	**54.10%**	**64.33%**	**27.03%**	**0.33**

class. ROS involves randomly duplicating examples from the minority class to balance the dataset. Table 14 presents a comparative analysis of these techniques.

As seen in the table, RUS consistently yields better results than the other balancing methods in all evaluation metrics. For example, using RUS relatively improves the localization performance in the Acc@1 metric by 17% and 36% compared to NCR and Tomek Links, respectively, and by 27% compared to ROS. These findings highlight RUS's superior ability to migate the effects of imbalance and contribute better localization performance. Based on these results, we choose RUS as the data balancing method for KBL to enhance the performance in the bug localization task.

In addition to examining the impact of different data balancing techniques on the localization performance of KBL, we further explore how varying the sampling ratio in RUS affects the performance of KBL. As shown in Table 13, the ratio of *non-keywords* to *keywords* in

Table 17 Query Improvement by KBL over Typical Strategies

Project	Query Pair	Improved	Worsened	Preserved
AspectJ	KBL vs. Title	60.34%	24.14%	15.52%
	KBL vs. Description	63.79%	18.97%	17.24%
	KBL vs. Title + Description	47.41%	25.86%	26.72%
Birt	KBL vs. Title	56.46%	19.31%	24.23%
	KBL vs. Description	82.16%	11.81%	6.03%
	KBL vs. Title + Description	53.14%	20.17%	26.69%
Platform. UI	KBL vs. Title	45.04%	29.82%	25.14%
	KBL vs. Description	70.73%	16.08%	13.19%
	KBL vs. Title + Description	42.00%	22.79%	35.21%
JDT	KBL vs. Title	41.30%	26.62%	32.08%
	KBL vs. Description	69.04%	15.48%	15.48%
	KBL vs. Title + Description	36.89%	23.66%	39.45%
SWT	KBL vs. Title	50.85%	21.90%	27.25%
	KBL vs. Description	71.17%	13.50%	15.33%
	KBL vs. Title + Description	41.48%	22.26%	36.25%
Tomcat	KBL vs. Title	45.37%	14.63%	40.00%
	KBL vs. Description	75.61%	11.71%	12.68%
	KBL vs. Title + Description	40.00%	16.10%	43.90%
All	KBL vs. Title	49.89%	22.74%	27.37%
	KBL vs. Description	72.08%	14.59%	13.33%
	KBL vs. Title + Description	43.49%	21.81%	34.70%

the six projects is less than 5. Therefore, we evaluate four sampling ratios, that is 4:1, 3:1, 2:1, and 1:1. Here N:1 indicates that the number of tokens labeled as *non-key* is *N* times the number of tokens labeled as *key* after sampling. Table 15 shows the performance of KBL when using RUS with different ratios. For comparison, the performance without balancing the original data is also provided. Note that, the 4:1 ratio only applies to AspectJ, Birt, and SWT, as only these projects present a > 4 ratio of the *non-keywords* and *keywords* before applying RUS. As shown in the table, KBL achieves the best performance with an RUS sampling ratio of 1:1, with all results being statistically significant at a p-value of 0.05. We also observe that as the sampling ratio increases, the performance of KBL tends to decline. For example, compared to using sampling ratios of 2:1, 3:1, 4:1, or no balancing, using the 1:1 sampling ratio improves KBL's localization performance in the Acc@1 metric by 7%, 47%, 71%, and 72%, respectively. This indicates that the bias introduced by the disparity between *non-keywords* and *keywords* becomes more pronounced as the difference grows, negatively affecting KBL's performance.

> KBL achieves better performance with RUS data balancing compared to the other three balancing techniques. When the number of non-keywords is balanced to match the number of keywords in training (with a 1:1 ratio), KBL outperforms all other sampling ratios.

5.3 RQ3 How does KBL Perform Compared to Traditional And Advanced Reformulation Approaches in Bug Localization?

5.3.1 RQ3.1 Does KBL Outperform the Typical Reformulation Strategies?

It is common for existing studies to directly use the title, the description, or both as the content proxy of a whole bug report to do bug localization. As a reformulation technique that mainly retrieves localization-hinting keywords from the textual content of the title and description, it is essential for us to check whether KBL could outperform these typical reformulation strategies. For each bug report, we retrieve its title and description items and then feed the title, the description, and both items to the Lucene search engine to retrieve buggy code files from the codebase separately. Then, we compare the retrieved results of these three typical strategies and that of KBL regarding Accuracy@K, MAP, and MRR respectively.

Table 16 shows the comparison results. From the table, we can observe that our reformulated queries are more effective than the typical queries (i.e., only using the title, the description, and both). In the six projects, KBL outperforms the *Title*, the *Description*, and the *Title + Description*. For example, for AspectJ, the buggy files of 25.86% bugs are returned as the top-1, while using typical strategies, only 11.20%, 12.93% and 18.10% bugs have their relevant files returned as top-1, respectively. Moreover, KBL achieves bug localization in 64.33% of the entire dataset consisting of 4,484 bug reports, with a mean average precision of 27.03% and a mean reciprocal rank of 0.33. These values are 6%, 9%, and 10% higher, respectively, than the performance of the *Title + Description* strategy.

For each project, we delve deeper into KBL's performance relative to the strategies in terms of the ranks of the first truly buggy code files of all bug reports. We compare our

Table 18 Comparison with Query Reformulation-Based Bug Localization Techniques

Project	Technique	Acc@1	Acc@5	Acc@10	Acc@20	MAP	MRR
AspectJ	TextRank	10.34%	27.58%	35.34%	49.13%	16.68%	0.19
	BLIZZARD	24.13%	**43.96%**	**51.72%**	**56.89%**	**28.18%**	**0.33**
	KBL	**25.86%**	37.06%	49.13%	**56.89%**	27.04%	0.32
Birt	TextRank	8.11%	19.31%	26.07%	34.56%	10.57%	0.14
	BLIZZARD	6.76%	15.86%	19.68%	26.07%	9.00%	0.11
	KBL	**12.30%**	**25.58%**	**33.70%**	**42.92%**	**14.66%**	**0.19**
Platform.UI	TextRank	13.73%	33.17%	43.09%	52.92%	19.66%	0.24
	BLIZZARD	10.30%	19.67%	24.98%	29.97%	12.67%	0.15
	KBL	**20.92%**	**42.85%**	**53.00%**	**62.99%**	**26.56%**	**0.32**
JDT	TextRank	17.96%	41.94%	52.28%	62.30%	23.87%	0.29
	BLIZZARD	14.59%	31.99%	40.49%	48.27%	18.35%	0.23
	KBL	**25.66%**	**52.12%**	**63.03%**	**73.45%**	**31.21%**	**0.38**
SWT	TextRank	17.51%	37.10%	49.02%	61.92%	23.68%	0.28
	BLIZZARD	21.41%	47.68%	58.39%	**72.50%**	29.11%	0.34
	KBL	**24.69%**	**48.41%**	**59.24%**	71.16%	**30.09%**	**0.36**
Tomcat	TextRank	29.26%	55.12%	67.31%	76.58%	35.34%	0.41
	BLIZZARD	36.58%	**65.36%**	**74.14%**	**82.43%**	**42.14%**	**0.49**
	KBL	**38.53%**	61.46%	69.75%	79.02%	41.34%	0.48
All	TextRank	15.20%	34.67%	44.55%	54.83%	20.56%	0.25
	BLIZZARD	14.45%	30.26%	37.39%	45.24%	18.35%	0.22
	KBL	**22.30%**	**44.02%**	**54.10%**	**64.33%**	**27.03%**	**0.33**

method with three strategies— *Title*, *Description* and *Title* + *Description*. If the rank of the first truly buggy file returned by KBL is higher than that of the typical strategy, we label it *query improvement*; conversely, if the rank is lower, we call it *query worsening*; otherwise, if their ranks are the same, we say it *query preserving*. The improvement, worsening, and preserving ratios for six projects are presented in Table 17. It can be observed that, compared to the *Description* strategy, there is an improvement in over 70% of queries in four projects. Across all six projects, there is an improvement in the effectiveness of over 60% queries, and the average proportion of queries showing improvement across all six projects reaches 72%. Compared with the *Title* strategy, we can see that, the proportion of the query improvements exceeds 50% in three projects, and the query improvement surpasses 40% in all six project systems. Moreover, in four projects, the improvement ratio surpasses the worsening ratios by a factor of more than two, and the average improvement ratio across all six systems is more than twice the worsening ratio. Although the improvement ratio in our method compared to the *Title* + *Description* strategy is not as high as that compared to the *Title* strategy and *Description* strategy, there is still one project with an improvement ratio

Table 19 Basic Statistics on the Number of Terms for KBL Queries (Q: quartile)

Project	Min Length	Max Length	Q1	Q2	Q3	Average Length
AspectJ	1	112	9	25	44	30
Birt	2	222	6	10	23	20
Platform_UI	2	412	9	16	32	38
JDT	2	384	10	20	36.5	38
SWT	1	222	7	17	32	25
Tomcat	2	122	5	8	16	15

exceeding 50%, four projects with improvement ratios exceeding 40%, and all six projects with improvement ratios exceeding 35%. The improvements validates that our KBL is very effective compared to typical reformulation strategies.

> KBL achieved much better performance than typical query reformulation strategies in terms of Accuracy@K, MAP, MRR, and effectiveness scores, with relative improvements of 3%-110% on Acc@1, 9%-93% on MAP, 10%-94% on MRR, and improving the effectiveness of 43% to 72% queries across six projects.

5.3.2 RQ3.2 Does KBL Perform Better Than The State-of-the-Art Reformulation Approaches?

To further validate the effectiveness of KBL, we compare KBL with two strong query reformulation techniques targeting feature/concept/bug location. We first compare with TextRank, a text-ranking algorithm that not only serves as a standalone query reformulation technique, but also acts as a crucial component in many other query reformulation frameworks (Rahman and Roy 2017; Kim and Lee 2019a; Rahman and Roy 2018b). It is a graph-based ranking model and it scores candidate keywords using word co-occurrences determined by a sliding window. After we obtain the content of the title and description of each bug report, we build a graph where a vertice is a term, an edge between two vertices is added if two terms co-occur within a sliding window (we set window size=2, a recommended value by Mihalcea and Tarau 2004). Once the text graph is constructed, we apply TextRank to estimate the importance of each term, and select the top 10 (according to previous practices (Rahman and Roy 2017; Kim and Lee 2019b)) terms as the search terms.

Besides TextRank, we also use BLIZZARD, which is a state-of-the-art query reformulation-based bug localization approach (with the replication package available) (Rahman and Roy 2018b), to evaluate the usability of our work. BLIZZARD considers the quality of bug reports by categorizing them into three types based on the content they contain and applying different reformulation strategies to them so as to enhance bug localization

Table 20 The Bug Localization Performance of KBL and TextRank when Both Select Top-10 Terms for Each Query

Project	Technique	Acc@1	Acc@5	Acc@10	Acc@20	MAP	MRR
AspectJ	TextRank	10.34%	27.58%	35.34%	**49.13%**	16.68%	0.19
	KBL_{10}	**15.51%**	**28.44%**	**37.06%**	45.68%	**17.79%**	**0.22**
Birt	TextRank	8.11%	19.31%	26.07%	34.56%	10.57%	0.14
	KBL_{10}	**8.73%**	**19.55%**	**27.67%**	**37.39%**	**11.24%**	**0.15**
Platform.UI	TextRank	13.73%	**33.17%**	**43.09%**	52.92%	19.66%	0.24
	KBL_{10}	**16.39%**	33.09%	**43.09%**	**53.47%**	**21.02%**	**0.25**
JDT	TextRank	17.96%	**41.94%**	**52.28%**	**62.30%**	**23.87%**	**0.29**
	KBL_{10}	**18.12%**	39.29%	49.31%	58.78%	22.67%	0.28
SWT	TextRank	17.51%	37.10%	49.02%	61.92%	23.68%	0.28
	KBL_{10}	**18.00%**	**43.06%**	**52.06%**	**62.04%**	**26.35%**	**0.31**
Tomcat	TextRank	29.26%	55.12%	**67.31%**	**76.58%**	35.34%	0.41
	KBL_{10}	**32.19%**	**57.56%**	66.82%	74.63%	**36.89%**	**0.43**
All	TextRank	15.20%	34.67%	44.55%	**54.83%**	20.56%	0.25
	KBL_{10}	**16.48%**	**35.14%**	**44.60%**	54.46%	**21.32%**	**0.26**

Table 21 The Bug Localization Performance of KBL and TextRank when Aligning the Term Count of TextRank with KBL for Each Query

Project	Technique	Acc@1	Acc@5	Acc@10	Acc@20	MAP	MRR
AspectJ	$TextRank_{align}$	15.51%	28.44%	38.79%	50.00%	20.70%	0.23
	KBL	**25.86%**	**37.06%**	**49.13%**	**56.89%**	**27.04%**	**0.32**
Birt	$TextRank_{align}$	8.85%	18.94%	26.44%	34.93%	11.08%	0.15
	KBL	**12.30%**	**25.58%**	**33.70%**	**42.92%**	**14.66%**	**0.19**
Platform.UI	$TextRank_{align}$	18.03%	37.47%	49.33%	59.17%	23.40%	0.28
	KBL	**20.92%**	**42.85%**	**53.00%**	**62.99%**	**26.56%**	**0.32**
JDT	$TextRank_{align}$	22.05%	47.63%	57.65%	68.32%	27.64%	0.34
	KBL	**25.66%**	**52.12%**	**63.03%**	**73.45%**	**31.21%**	**0.38**
SWT	$TextRank_{align}$	18.73%	38.32%	50.60%	62.16%	24.84%	0.29
	KBL	**24.69%**	**48.41%**	**59.24%**	**71.16%**	**30.09%**	**0.36**
Tomcat	$TextRank_{align}$	26.82%	55.12%	67.80%	74.63%	33.88%	0.40
	KBL	**38.53%**	**61.46%**	**69.75%**	**79.02%**	**41.34%**	**0.48**
All	$TextRank_{align}$	17.95%	37.73%	48.28%	58.34%	23.01%	0.30
	KBL	**22.30%**	**44.02%**	**54.10%**	**64.33%**	**27.03%**	**0.33**

performance. While newer techniques have emerged since BLIZZARD's introduction in 2018, most (e.g., Kim et al. 2021, Chaparro et al. 2019) tend to focus on specific enhancements rather than presenting fundamentally novel breakthroughs like BLIZZARD. Consequently, BLIZZARD remains representative among query reformulation techniques and is frequently used as a baseline in evaluating new IRBL methods (Shao and Yu 2023; Li et al. 2021), underscoring its continued relevance in the field.

Table 18 shows the comparative results of our method with TextRank and BLIZZARD. In the table, we can find that TextRank performed less effectively than BLIZZARD on AspectJ, SWT, and Tomcat, but better in the remaining three projects. By checking the overall performance (i.e., putting bug reports of six projects together as a dataset) shown in the last row of Table 18, TextRank is found to achieve better performance than BLIZZARD. Further, we can find that KBL performs better than TextRank in all six projects. KBL provides relatively 47% higher Acc@1, 27% higher Acc@5, 21% higher Acc@10, 17% higher Acc@20, 31% higher MAP and 32% higher MRR than TextRank for all six projects. As for BLIZZARD, we can see that KBL greatly outperforms BLIZZARD in four out of six projects except for AspectJ and Tomcat regarding all performance metrics. For instance, compared to Birt, KBL shows relative improvements of 82%, 61%, 71%, and 65% in Acc@1, Acc@5, Acc@10, and Acc@20 metrics, respectively. Additionally, KBL achieves a 63% relative improvement in MAP and a 73% relative improvement in MRR. For AspectJ and Tomcat, KBL and BLIZZARD have similar performance in MAP and MRR. KBL only outperforms BLIZZARD relatively by 7% and 5% in Acc@1. In Acc@5,10,15 and 20, BLIZZARD is much better than KBL. In other words, our keywords-based reformulation approach is able to obtain better localization performance than the state-of-the-art reformulation strategies.

Table 19 further presents the statistics on the length of reformulated queries generated by KBL across different projects. As shown, the median length (i.e., the Q2 column) across the six projects is 25, 10, 16, 20, 17, and 8, respectively, while the average query length is 30, 20, 38, 38, 25, and 15. These results indicate that KBL dynamically adjusts the length of the reformulated queries based on the specific characteristics of each bug report. Similarly, BLIZZARD reformulates queries based on the type of bug report, resulting in variable

query lengths rather than a fixed length for all bug reports. In contrast, as a default setting, TextRank consistently selects the top-10 highest-scoring terms for each reformulation, resulting in a fixed query length each time.

Given the differences in query length between KBL and TextRank, we conduct additional experiments to provide a fair comparison with TextRank. Specifically, we limit KBL to the top-10 terms by probability score for query reformulation (denoted as KBL_{10}) and compare it with TextRank. Additionally, we adjust TextRank to output the same number of terms as KBL for each query, referred to as $TextRank_{align}$. For example, if KBL outputs N terms for a bug report, TextRank is adjusted to output its top-N terms for the same bug report. Tables 20 and 21 present the comparison results between TextRank and KBL_{10}, and between $TextRank_{align}$ and KBL, respectively.

As shown in Table 20, KBL_{10} outperforms TextRank across most evaluation metrics in five out of six projects, with the exception of the JDT project, where TextRank achieves better results. This suggests KBL_{10} generally provides more effective query for bug localization compared to TextRank in the majority of cases. For example, in the SWT project, KBL_{10} achieves relative improvements over TextRank of 3%, 16%, and 6% in Acc@1, Acc@5, and Acc@10, respectively. KBL_{10} also shows relative improvements of 11% and 10% in MAP and MRR. Combining the results across all six projects, we observe that KBL_{10} outperforms TextRank in all metrics except for Acc@20, indicating that KBL_{10} is still slightly superior to TextRank overall.

As seen in Table 21, KBL consistently outperforms $TextRank_{align}$ across all evaluation metrics, showing a substantial performance advantage. Specifically, across all six projects, KBL shows notable relative improvements over $TextRank_{align}$, with increases of 24%, 16%, 12%, 10%, 17%, and 10% in the Acc@1, Acc@5, Acc@10, Acc@20, MAP, and MRR, respectively. These improvements highlight KBL's stronger capability to capture and utilize key information of bug reports, resulting in more accurate localization outcomes.

> KBL is found to exhibit relatively, 47% higher MAP and 50% higher MRR than BLIZZARD, 32% higher MAP and 32% MRR than TextRank in six projects; KBL outperforms TextRank in all six projects and performs better than BLIZZARD in most cases at different accuracy@K. When the query length of KBL is limited to 10 terms like default TextRank, its performance slightly surpasses that of TextRank. However, when the query length of TextRank is adjusted to match that of KBL, the performance of KBL significantly exceeds that of TextRank.

5.4 RQ4. Could Queries Generated by KBL Further Enhance the Localization Performance of Representative IRBL Techniques?

To thoroughly assess the effectiveness of our KBL, we further combine KBL into seven advanced IRBL techniques, including BugLocator (Zhou et al. 2012), BRTracer (Wong et al. 2014), Locus (Wen et al. 2016), BLIA (Youm et al. 2015), BLUiR (Saha et al. 2013), Amalgam (Wang and Lo 2014), and D&C (Koyuncu et al. 2019). BugLocator (Zhou et al. 2012) utilizes a revised Vector Space Model (rVSM) and leverages information from historically similar bug reports to do bug localization. BRTracer (Wong et al. 2014) extends BugLocator by incorporating stack trace from bug reports and source file segmentation to improve the retrieval of buggy code files. Locus (Wen et al. 2016) uses textual and supple-

Table 22 Performance Comparison of BugLocator and $\text{BugLocator}_{\text{KBL}}$

Model	Acc@1	Acc@5	Acc@10	Acc@20	MAP	MRR
BugLocator	20.11%	40.09%	50.04%	59.34%	24.13%	0.29
$\text{BugLocator}_{\text{KBL}}$	**23.21%**	**46.11%**	**55.19%**	**65.56%**	**28.51%**	**0.34**

Table 23 Performance Comparison of BRTracer and $\text{BRTracer}_{\text{KBL}}$

Model	Acc@1	Acc@5	Acc@10	Acc@20	MAP	MRR
BRTracer	23.19%	45.33%	54.90%	65.41%	28.67%	0.34
$\text{BRTracer}_{\text{KBL}}$	**25.91%**	**48.66%**	**57.87%**	**69.13%**	**32.49%**	**0.38**

Table 24 Performance Comparison of Locus and $\text{Locus}_{\text{KBL}}$

Model	Acc@1	Acc@5	Acc@10	Acc@20	MAP	MRR
Locus	24.50%	46.03%	56.08%	66.36%	30.35%	0.35
$\text{Locus}_{\text{KBL}}$	**26.36%**	**48.97%**	**58.47%**	**69.24%**	**33.33%**	**0.39**

Table 25 Performance Comparison of BLIA and BLIA_{KBL}

Model	Acc@1	Acc@5	Acc@10	Acc@20	MAP	MRR
BLIA	21.63%	42.12%	51.13%	60.95%	26.01%	0.30
BLIA_{KBL}	**24.75%**	**48.84%**	**56.91%**	**67.17%**	**30.51%**	**0.36**

mentary information from historical code changes to identify buggy code files in response to the given bug report. BLIA (Youm et al. 2015) integrates various types of information, such as stack traces, method names, similarity between bug reports and method names, and comments from bug reports, to enhance bug localization performance. BLUiR (Saha et al. 2013) divides bug reports into summary and description, and extracts structured information like class names, variable names, and comments from source code files to improve matching accuracy. Amalgam (Wang and Lo 2014) integrates BugLocator and BLUiR, along with version history analysis, to improve bug localization. Proposed by Koyuncu et al. (2019), D&C employs a gradient boosting supervised learning approach for bug localization, focusing on feature types that perform well across specific bug report collections. They train multiple classifiers based on the strengths of various localization tools, assigning optimal weights to similarity measurements between bug reports and source code files.

In this study, BugLocator, BRTracer, Locus, BLIA, BLUiR, and Amalgam are implemented by directly using the replication packages provided by Bench4BL (Lee et al. 2018), while D&C is implemented using the open-source code[9] provided by its authors. We combine KBL with the seven IRBL techniques by replacing the original bug report text used as input in these IRBL techniques with the query reformulated by KBL. The specific integration methods for each IRBL technique are described in detail as follows.

$\text{BugLocator}_{\text{KBL}}$ To integrate BugLocator with KBL, we replace the original bug report text with the keywords generated by KBL, using this reformulated text as input for BugLocator. For previously fixed bug reports, we replace their text with the golden keywords we constructed.

[9] https://github.com/TruX-DTF/d-and-c

Table 26 Performance Comparison of BLUiR and $BLUiR_{KBL}$

Model	Acc@1	Acc@5	Acc@10	Acc@20	MAP	MRR
BLUiR	15.90%	30.99%	41.61%	52.02%	19.15%	0.24
$BLUiR_{KBL}$	**21.61%**	**40.99%**	**51.24%**	**62.77%**	**25.40%**	**0.30**

Table 27 Performance Comparison of Amalgam and $Amalgam_{KBL}$

Model	Acc@1	Acc@5	Acc@10	Acc@20	MAP	MRR
Amalgam	15.96%	31.53%	42.79%	53.23%	19.48%	0.24
$Amalgam_{KBL}$	**21.21%**	**41.07%**	**50.82%**	**61.77%**	**25.66%**	**0.30**

Table 28 Performance Comparison of D&C and $D\&C_{KBL}$

Model	Acc@1	Acc@5	Acc@10	Acc@20	MAP	MRR
D&C	26.31%	49.57%	57.87%	68.42%	32.05%	0.38
$D\&C_{KBL}$	**29.17%**	**52.38%**	**61.30%**	**71.72%**	**35.54%**	**0.41**

$BRTracer_{KBL}$ Similar to the approach for integrating BugLocator with KBL, when combining BRTracer with KBL, we replace the bug report text used in BRTracer with the keywords generated by KBL. Specifically, we use these keywords in place of the original bug report text for calculating rVSMseg and SimiScore. Likewise, in line with the approach used in $BugLocator_{KBL}$, we replace historical bug report texts with golden keywords.

$Locus_{KBL}$ In Locus, bug reports are primarily used to construct natural language (NL) queries and code entity (CE) queries. When integrating KBL with Locus, we replace the original bug report text with the keywords output by KBL, using them as the NL query for the bug report.

$BLIA_{KBL}$ In BLIA, the bug report text is divided into two parts, the summary and the description, both used to calculate StructVsmScore. When integrating KBL with BLIA, we first determine whether the keywords identified by KBL originate from the summary or the description. We then replace the original summary with the keywords found in the summary, and replace the original description with the keywords found in the description. If no keywords appear in either the summary or the description, we retain the original text for that part of the bug report to ensure completeness. Additionally, for calculating the SimiBugScore, we also replace the text of historical bug reports with golden keywords, as done in $BugLocator_{KBL}$.

$BLUiR_{KBL}$ To integrate BLUiR with KBL, we replace the summary and the description of the bug report separately with keywords output by KBL, following the approach used in $BLIA_{KBL}$.

$Amalgam_{KBL}$ We primarily integrate KBL into the Similar Report Component and Structure Component of Amalgam. Specifically, in the Similar Report Component, we replace the text of the new bug report with keywords output by KBL, and substitute historical bug report texts with golden keywords. In the Structure Component, similar to the $BLIA_{KBL}$, we replace the keywords separately in the bug report's summary and description.

D&C$_{KBL}$ When integrating D&C with KBL, we apply KBL-generated keywords for three feature extraction, keywords from the summary are used for the *summary* feature, keywords from the description are used for the *description* feature, and all keywords are used for the *rawBugReport* feature. Additionally, we replace the original six IR techniques, namely BugLocator, BRTracer, Locus, BLIA, BLUiR, and Amalgam, with their KBL-integrated versions, that is BugLocator$_{KBL}$, BRTracer$_{KBL}$, Locus$_{KBL}$, BLIA$_{KBL}$, BLUiR$_{KBL}$, and Amalgam$_{KBL}$.

Tables 22, 23, 24, 25, 26, 27, and 28 show the comparison results between the original seven IRBL techniques and their KBL-integrated variants. As shown in Table 22, BugLocator combined with KBL demonstrates notable performance improvements across all metrics compared to the original BugLocator. Specifically, in the Acc@1, Acc@5, Acc@10, and Acc@20 metrics, BugLocator$_{KBL}$ achieves relative improvements of 15%, 15%, 10%, and 10%, respectively. Furthermore, BugLocator$_{KBL}$ also shows notable gains in ranking-based metrics, with relative improvements of 18% in MAP and 17% in MRR over the original BugLocator. These results indicate that the integration of KBL enhances BugLocator's effectiveness in accurately localizing buggy code files.

Table 23 presents the performance comparison between BRTracer and BRTracer$_{KBL}$ in the bug localization task. It can be observed that BRTracer, when combined with KBL, achieves relative improvements of 12%, 7%, 5%, 6%, 13%, and 12% over the original BRTracer in the Acc@1, Acc@5, Acc@10, Acc@20, MAP and MRR metrics, respectively. These improvements suggest that the integration of KBL helps BRTracer perform better in identifying and locating buggy code, resulting in a more effective bug localization process.

Table 24 presents a comparison of the bug localization performance between Locus and Locus$_{KBL}$. The results indicate that Locus$_{KBL}$ outperforms the original Locus in all evaluation metrics. Specifically, Locus$_{KBL}$ demonstrates relative improvements of 8%, 6%, 4%, 4%, 10%, and 11% in the Acc@1, Acc@5, Acc@10, Acc@20, MAP, and MRR metrics, respectivel. These improvements reflect the positive impact of integrating KBL on Locus's effectiveness in bug localization.

In Table 25, we present the bug localization performance comparison between BLIA and BLIA$_{KBL}$. It is evident that integrating KBL enhances the overall performance of BLIA. Specifically, BLIA$_{KBL}$ demonstrates relative improvements of 14%, 16%, 11%, 10%, 16%, and 20% in the Acc@1, Acc@5, Acc@10, Acc@20, MAP, and MRR metrics, respectively, compared to the original BLIA. These improvements indicate that KBL contributes to more effective bug localization, making BLIA better equipped to accurately identify relevant buggy code files.

Table 26 presents a comparison of the localization performance between BLUiR$_{KBL}$ and BLUiR, showing that BLUiR$_{KBL}$ significantly enhances bug localization effectiveness. The integration of KBL allows BLUiR to more accurately identify buggy code files related to bug reports. Notably, BLUiR$_{KBL}$ achieves relative improvements of 36%, 32%, 24%, 21%, 33%, and 25% in Acc@1, Acc@5, Acc@10, Acc@20, MAP, and MRR, respectively, compared to BLUiR.

Table 27 illustrates the localization performance between Amalgam and Amalgam$_{KBL}$, demonstrating that Amalgam$_{KBL}$ consistently outperforms the original Amalgam across all evaluation metrics. This highlights the positive impact of integrating KBL into Amalgam, significantly enhancing its bug localization effectiveness. In particular, Amalgam$_{KBL}$ achieves

relative improvements of 25%, 30%, 19%, and 16% for the Acc@1, Acc@5, Acc@10, and Acc@20 metrics, respectively. Moreover, $\text{Amalgam}_{\text{KBL}}$ shows relative improvements of 32% and 25% in the MAP and MRR metrics compared to Amalgam, further confirming the benefits of incorporating KBL for more accurate localization of buggy code files.

The above experimental results indicate that KBL enhances the performance of six widely-used IRBL techniques to varying degrees. To further explore whether KBL can similarly benefit the more advanced IRBL technique (i.e., D&C, an ensemble approach of six IRBL techniques), we compare the bug localization performance of D&C and $\text{D\&C}_{\text{KBL}}$. The comparison results are presented in Table 28. The results show that D&C combined with KBL consistently outperforms D&C alone, confirming that KBL can also effectively improve the performance of D&C. Specifically, $\text{D\&C}_{\text{KBL}}$ achieves relative improvements of 11%, 6%, 6%, and 5% in Acc@1, Acc@5, Acc@10, and Acc@20, respectively, compared to D&C. Additionally, $\text{D\&C}_{\text{KBL}}$ shows relative improvements of 11% in MAP and 8% in MRR, further emphasizing the positive impact of integrating KBL for more accurate bug localization.

Overall, these findings demonstrate that the proposed KBL effectively enhances existing IRBL techniques, resulting in recognizable improvements in the bug localization task. This further highlights the potential of KBL as a valuable reformulation strategy for broader applications in bug localization, showcasing its capacity to contribute to performance gains across various IRBL techniques.

> Using the reformulated queries generated by KBL, the performance of seven representative IRBL techniques shows recognizable improvements, including relative increases of 8%-36% in Acc@1, 6%-32% in Acc@5, 4%-24% in Acc@10, 4%-21% in Acc@20, 10%-33% in MAP, and 8%-25% in MRR. These results further confirm the effectiveness of KBL and highlight its considerable potential for improving bug localization.

5.5 Statistical Significance Tests Over Observed Performance Differences

In the above four RQs, we observed different kinds of performance differences between certain two approaches. This section mainly aims to do statistical significance tests to check whether these observed differences have statistical significance or not. With such tests, the effectiveness and potential of our KBL in bug localization could be better understood. Specifically, Wilcoxon Rank Sum test (Mann and Whitney 1947) and Cliff's Delta effectsize (Macbeth et al. 2011) are used to perform this task. Wilcoxon Rank Sum test is a nonparametric statistical test commonly used to compare two groups of non-parametric interval or not normally distributed data. It can tell us whether statistical significance exists or not. With the help of Cliff's delta effect size, we can measure how large the difference might be quantitatively.

For those experiments involving comparing KBL (or its components) with baselines in the above four RQs (i.e., Sections 5.1 to 5.4), we perform corresponding statistical tests using the Wilcoxon Rank Sum test and Cliff's Delta effect size. Due to the relatively large number of such need-to-be-conducted tests, which take up considerable space, we choose to place the testing results on GitHub[10] instead of including them in the main text. The

[10] https://github.com/Caiby0927/KBL

obtained testing results are all or, in most cases, have statistical significance at the p-value of 0.05, with different degrees of effect sizes. This means, the major conclusions arrived in Sections 5.1 to 5.4 still holds based on our testing results.

6 Discussion

The experimental results in Section 5 have validated the effectiveness of our KBL in bug localization tasks. In this section, we mainly discuss some potential design improvements of our KBL, the practical considerations in real-world adoption of KBL, and the potential threats to our study.

6.1 Potential Design Improvements of KBL

Addressing Potential Local Optima in Keyword Selection In our study, we utilized a genetic algorithm (GA) for keyword selection. Despite the constructed keyword benchmark having proved to be rather effective in locating bugs (in RQ1), they may still not be the best benchmark but the suboptimal one, as theoretically speaking, the GA may yield results that are local optima rather than global optima. It would be valuable to explore possible strategies that help GA to generate global optimal results as much as possible. For example, future research could try to combine GA with other optimization techniques such as simulated annealing, to improve its ability to explore the solution space more thoroughly. Or, design suitable multiple objective criteria to make GA generate more diverse initial populations and explore the use of dynamic mutation rate to maintain and even enhance population diversity throughout the GA evolution process.

Improving Noisy Terms Identification In our current approach, we utilize heuristic rules to filter out noisy terms by comparing the retrieval performance of two consecutive subsentences. While this method effectively identifies terms likely to be noise based on performance discrepancies, it is possible that some retained keywords may still be noisy. Towards this, future research could try to incorporate other useful measures like leveraging clustering algorithms to group similar terms and use them to help identify outliers that could be flagged as noise. Or, integrating suitable machine learning models that require minimal labeled data to help predict the likelihood of noise for each term based on learned patterns. Or, not relying solely on the performance discrepancies, but comprehensively taking several metrics like contextual semantic alignment within bug reports or the domain specificity of a term in the field of bug localization.

Enhancing Keyword Classification Performance Keyword classifier is a key component of our KBL. The reformulation process is based on the keywords predicted by the keyword classifier. According to the performance table of our keyword classifier, we can see that the current performance is still quite modest, far from satisfactory. Despite we have made various attempts to improve classification accuracy, including testing different sampling methods, training the classifier with the fine-tuned large language model, integrating features extracted from the large language model with existing 61 features, performing feature importance analysis to remove low-importance features, further refining the golden key-

words by removing more general terms, and experimenting with more complex neural network models for classification (the results are available in our repository[11]), the results have not yet surpassed the reported performance of KBL in this paper. From another perspective, the findings of our RQ3 and RQ4 indicated that, even based on the not-so-satisfactory keyword classification performance, our KBL still outperformed strong query reformulation baselines and enhanced typical IRBL techniques based on our reformulated queries. We believe that it is quite rewardable for more researchers to join us to address the keyword prediction problem based on the already-constructed golden keyword benchmark by us.

6.2 Practical Considerations in Real-World Adoption of KBL

Adaption to Different Types of Bug Reports and Localization Scenarios Currently, KBL is not specifically designed to particular types of bug reports or specific localization scenarios. In other words, it serves as a general-purpose reformulation technique for bug reports from open-source projects without distinguishing between report types. Theoretically, KBL can be easily adapted to specific bug reports in both open-source and closed-source projects, as long as the artifacts required—source/bug-fixing code and the textual problem description of bug reports—are available in real-world software projects.

However, we acknowledge that the features we designed for KBL's keyword classifier may not be optimal for all types of bug reports. For example, security bug reports may contain unique characteristics, such as vulnerability patterns, which could be highly effective in identifying keywords for bug localization. In such cases, adapting the keyword classification model to incorporate specialized features tailored to specific bug report types could potentially enhance its performance. Additionally, several strategies could also be referred to, to enhance KBL's ability to process diverse bug reports without requiring specialized adaptations to different contexts. One is to leverage feature learning through machine learning or deep learning models, allowing the keywords classification component of KBL to automatically learn patterns from bug reports across various domains. Another one is to replace the keywords classification component with pre-trained large language models and introduce domain-specific fine-tuning for adaptation.

Data Availability and Time Cost of GA One potential challenge for real-world implementation of KBL, particularly in industry settings, is the requirement of a well-linked historical bug report dataset with corresponding buggy code. This is crucial because KBL relies on linked historical bug reports (knowing the buggy code files associated with each bug report) to generate the golden keywords and to further train the keyword classifier. For projects without their own or sufficient historical bug report data, a feasible solution is to train KBL using bug reports from other projects, such as open-source datasets, and then apply it to classify keywords in the current bug reports. While this cross-project application is a practical strategy, its effectiveness remains to be further verified, as the transferability of keyword classification across different projects may vary.

Another aspect that needs to be noticed is that the process of obtaining these golden keywords involves the use of a genetic algorithm, which does have some computational cost. However, since keyword extraction using GA needs only to be done once for a project

[11] https://github.com/Caiby0927/KBL

and could be conducted offline (i.e., the construction of the golden keyword benchmark can be done in advance during idle periods), we think the time costly problem would not be so significant in practical adoption of KBL. Once the keywords are constructed, KBL operates efficiently in real-time (as shown by Fig. 5 in Appendix A.2 Section).

6.3 Threats to Validity

Internal Validity There are mainly three threats to the internal validity of our study. One is that, we directly use a dataset shared by Ye et al. (2014) to construct our golden keywords benchmark. This dataset provides a number of bug reports and their associated buggy code files, and has been commonly used in existing IRBL studies (Lam et al. 2017; Shao and Yu 2023; Xiao et al. 2023, 2017). Still, we have to admit that any potential bias in linking bug reports with their buggy code may negatively affect our study.

One more threat arises from the fact that some bug reports in the dataset may actually be feature requests or other non-bug issues (Kochhar et al. 2014). In this study, we did not apply a filtering process to identify and exclude these reports, aside from eliminating bug reports that only involved code file additions for closing them (actually, the lack of ground truth of the shared datasets also makes it challenging for us to construct a pure dataset that contains only true positive bug reports). We acknowledge that this remains a potential limitation and should be considered when interpreting our findings.

Another threat lies in the comparison with the reformulation techniques, as we failed to find a replication package of the TextRank, we re-implement this technique based on its description with recommended settings and parameters (Mihalcea and Tarau 2004), any mistakes in implementation would threaten our arrived conclusions. To avoid this, we conduct several rounds of code reviews to make sure our re-implementations are correct.

External Validity The threats to external validity mainly lie in generalizing our findings to a broader range of software projects and specific types of bug reports. As with most empirical studies, our experiments were conducted on a limited number of datasets. Although we selected six widely studied open-source projects of different domains, scales, and complexities, these may not fully capture the diversity of real-world software systems, particularly in commercial or large-scale environments. Extending future evaluations to more diverse and representative projects remains a valuable effort to further strengthen the generalization of our KBL in real-world scenarios.

Besides, the features designed for KBL's keyword classifier may not generalize effectively to different types of bug reports. For instance,security bug reports may exhibit unique characteristics, like vulnerability patterns or exploit scenarios, that may not be fully captured by a generalized feature set. It is appreciated that future research explores adapting the keyword classification model with specialized features tailored to the target type of bug reports.

7 Conclusion

To mitigate the weakness of existing reformulation strategies, we propose to develop KBL, a golden keywords-guided query reformulation technique for bug localization. First, we combine genetic algorithms and summarize keywords refinement rules to construct a benchmark that contains keywords which perform quite well in locating bugs. Then, we use this benchmark to build a keywords classifier to identify keyword candidates from a bug report that may be good hints for bug localization. These candidates would go through noise removal and shared keywords expansion, the output of which would work as the final query used to retrieve buggy code for the bug report. The experimental results demonstrate that our constructed keywords benchmark is of high quality. The KBL is found to outperform existing reformulation strategies with substantial improvements and could advance representative IRBL techniques with noticeable enhancements to their performance. In the future, we plan to explore additional bug-related semantic features to enhance both the classification performance of the keywords classifier and the retrieval effectiveness of the reformulated queries. Furthermore, we also plan to apply our technique to more complex scenarios, such as bug auto-repair, to explore its applicability in broader contexts.

A Appendix

A.1 Confusion Matrices of Keywords Classifier Over Six Projects

Tables 29, 30, 31, 32, 33, and 34, present the detailed confusion matrices of keywords classifier for the six experimental projects.

Table 29 Confusion Matrix for the AspectJ Project

Labels	Key	Non-Key
Key	6,604	3,615
Non-Key	848	1,731

Table 30 Confusion Matrix for the Birt Project

Labels	Key	Non-Key
Key	36,141	9,083
Non-Key	6,322	5,393

Table 31 Confusion Matrix for the Eclipse_Platform_UI Project

Labels	Key	Non-Key
Key	109,087	35,539
Non-Key	13,715	18,013

Table 32 Confusion Matrix for the JDT Project

Labels	Key	Non-Key
Key	71,993	26,892
Non-Key	7,631	13,759

Table 33 Confusion Matrix for the SWT Project

Labels	Key	Non-Key
Key	33,211	13,582
Non-Key	3,650	7,245

Table 34 Confusion Matrix for the Tomcat Project

Labels	Key	Non-Key
Key	8,665	2,650
Non-Key	1,240	1,349

A.2 Time Cost of Extracting Features for Bug Reports

To better understand the practical use of our KBL, we also investigate the time required to extract term features from bug reports (required by keywords classifier). The experiment is conducted on a Windows-based machine equipped with an Intel i5-12600KF CPU and 64GB of RAM. Following the sampling strategy in Krejcie (1970), which ensures representativeness and reliability with a 95% confidence level and a 5% margin of error, we randomly sampled 384 bug reports from six projects (with an average length of 124.3 terms). We record the total feature extraction time for each report as well as the number of terms in each report. Our results show that the average time per term is approximately 0.4964 seconds. For a bug report containing around 100 terms, feature extraction would thus take roughly 49.64 seconds. In our experimental setup, the 61 features are calculated sequentially for each report, with each feature computation starting only after the previous one is completed. However, using distributed or parallel computing could significantly reduce the time cost, greatly enhancing the practical usability of our KBL. To better illustrate time distribution, we've included box plots depicting the time cost for these 384 bug reports in Fig. 5.

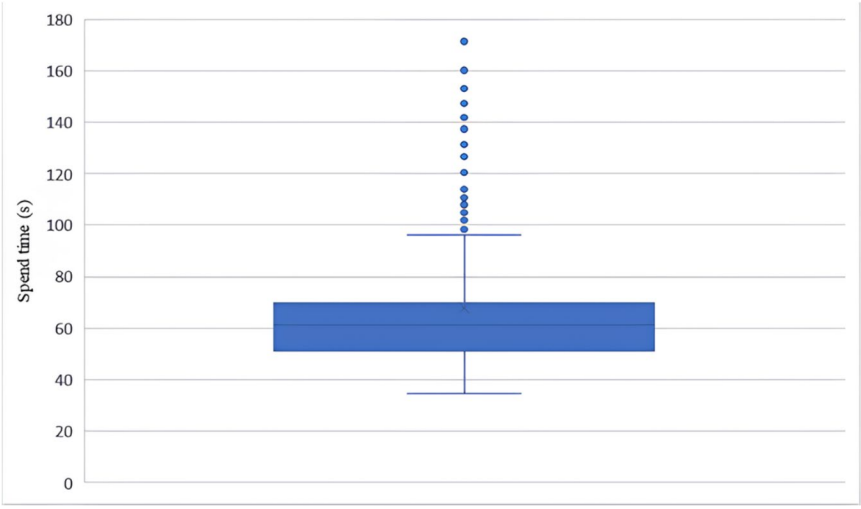


Fig. 5 The time cost distribution of feature extraction for sampled 384 bug reports

A.3 Performance Comparison Between Doing RUS Ten Times and Once

During keyword classifier building, we set a fixed sampling seed of 1,391 for RUS and applied RUS only once. To mitigate any potential biases introduced by RUS, we also repeat the RUS process 10 times without setting a sampling seed and average the experimental results. Table 35 shows their performance differences in terms of F1 score of keywords classifier and the final obtained Acc@K, MAP and MRR bug locating performance. From the table, we only find a slight performance difference. This indicates that the findings of our study are not significantly impacted by whether RUS is performed once or multiple times.

Table 35 The Keyword Classification Performance and Bug Localization Performance after Applying RUS Ten times (TenTimesAverage) or Once (KBL)

Project	Technique	F1	Acc@1	Acc@5	Acc@10	Acc@20	MAP	MRR
AspectJ	TenTimesAverage	0.44	24.48%	37.40%	47.41%	57.24%	27.30%	0.32
	KBL	0.45	25.86%	37.06%	49.19%	56.89%	27.04%	0.32
Birt	TenTimesAverage	0.42	11.79%	24.54%	33.11%	42.51%	14.31%	0.19
	KBL	0.42	12.30%	25.58%	33.70%	42.92%	14.66%	0.19
Platform.UI	TenTimesAverage	0.42	20.53%	42.05%	52.56%	62.53%	26.16%	0.31
	KBL	0.43	20.92%	42.85%	53.00%	62.99%	26.56%	0.32
JDT	TenTimesAverage	0.45	25.01%	52.04%	62.74%	73.20%	31.03%	0.38
	KBL	0.45	25.66%	52.12%	63.03%	73.45%	31.21%	0.38
SWT	TenTimesAverage	0.45	24.29%	47.38%	59.01%	70.84%	29.76%	0.35
	KBL	0.46	24.69%	48.41%	59.24%	71.16%	30.09%	0.36
Tomcat	TenTimesAverage	0.41	36.10%	61.03%	71.12%	79.66%	40.45%	0.47
	KBL	0.42	38.53%	61.46%	60.75%	79.02%	41.34%	0.48

A.4 Query Reformulation With or Without Keyword Expansion or Low-Quality Term Removal

The query reformulation module of KBL default applies keyword expansion and low-quality term removal to an initial query output by the keywords classifier. To understand the impact of these two reformulation strategies, we built two variants of the query reformulation module. They are as follows: (1) *without expansion*: in this variant, shared keywords are not used for expanding and augmenting a query. (2) *without noise removal*: in this variant, the low-quality term is not removed, and only performs the shared keywords expansion. After we obtain the corresponding queries for the above reformulation variants, we use them to retrieve buggy code files and do further results comparison. Table 36 shows the results. It reveals that the model KBL with both steps (i.e., applying shared keywords expansion and noise removal) demonstrates superior performance in bug localization compared to the two variants (i.e., without expansion, and without noise removal). We can observe that the performance of both variants decreases to some extent compared to KBL. Among them, the *without expansion* variant experiences a smaller decrease compared to KBL, with a relative decrease of 1% in acc@1 metric. On the other hand, the *without noise removal* variant shows a relatively larger decrease, with a 6% relative decrease in acc@1 metric.

Table 36 Bug Localization Performance of KBL with/without Keyword Expansion or Noise Removal

Model Varaint	Acc@1	Acc@5	Acc@10	Acc@20	MAP	MRR
without expansion	22.03%	43.53%	53.56%	64.07%	26.85%	0.32
without noise removal	21.05%	43.19%	53.18%	63.40%	26.40%	0.32
KBL	**22.30%**	**44.02%**	**54.10%**	**64.33%**	**27.03%**	**0.33**

A.5 The Settings of Similarity Threshold and Repetition Time

Similarity threshold is a key parameter used to filter similar historical bug reports for constructing shared keywords. We test four candidate values for the parameter *similarity threshold*: (0.6, 0.7, 0.8, 0.9). *Repetition time* is another key parameter used to limit the maximum number of occurrences of the same term in a query (belonging to the noise removal part). We test five value settings, i.e., (1, 2, 3, 4, 5) for this parameter. To better understand the individual impact of each parameter on localization performance, we assess how performance changes when adjusting each parameter independently. Specifically, when testing *Similarity threshold*, we vary its value from 0.6 to 0.9 in increments of 0.1, while keeping *repetition time* fixed at 4 (the optimal value in determined in our analysis). Similarly, when testing *repetition time*, we vary its value from 1 to 5 in increments of 1, with the *Similarity threshold* fixed at 0.6 (also identified as the optimal value). Table 37 and 38 present the localization performance of KBL under different settings for *similarity threshold* and *repetition time*, respectively.

Table 37 Bug Localization Performance of KBL with Different Similarity Threshold

Similarity Threshold	Acc@1	Acc@5	Acc@10	Acc@20
0.6	**(1000)22.30%**	**(1974)44.02%**	**(2426)54.10%**	**(2885)64.33%**
0.7	(990)22.07%	(1961)43.73%	(2420)53.96%	(2879)64.20%
0.8	(990)22.07%	(1959)43.68%	(2411)53.76%	(2878)64.18%
0.9	(993)22.14%	(1957)43.64%	(2407)53.67%	(2875)64.11%

From Table 37, we see that when the similarity threshold parameter increases from 0.6 to 0.9, the acc@1, acc@5, acc@10 and acc@20 metrics of the queries show a generally decreasing trend. For example, as the similarity threshold value increases from 0.6 to 0.9, the accuracy@10 metric for KBL decreases from 54.10% to 53.67%, and the number of successfully localized bugs decreases from 2426 to 2407. Meanwhile, we can also observe that the performance decrease is quite small when the similarity threshold is increased. This means KBL demonstrates a certain level of robustness to variations in the similarity threshold parameter (when set >=0.6).

From Table 38, we can observe that as the repetition time increases, the query performance initially improves and then declines. When the repetition time reaches 4, the query performance is optimal, while it is worst when the repetition time is 1. Compared to the optimal repetition time, queries with a repetition time of 1 experience a relatively 7% decrease in the acc@1 metric. This indicates that the query performance is sensitive to the repetition time of keywords, which however is not taken into consideration by TextRank and BLIZZARD.

Table 38 Bug Localization Performance of KBL with Different Repetition Time

Repetition Time	Acc@1	Acc@5	Acc@10	Acc@20
1	(934)20.82%	(1888)42.10%	(2358)52.58%	(2808)62.62%
2	(985)21.96%	(1958)43.66%	(2409)53.72%	(2891)64.47%
3	(994)22.16%	(1966)43.84%	(2434)54.28%	(2893)64.51%
4	**(1000)22.30%**	**(1974)44.02%**	**(2426)54.10%**	**(2885)64.38%**
5	(994)22.16%	(1966)43.84%	(2427)54.12%	(2879)64.20%

A.6 The Performance of Applying Golden Keywords Beyond Bug Localization Tasks

In this study, with the aim of providing good localization guidance for bug localization tasks, we constructed a golden keywords dataset based on historical bug-locating data and used it to build a keyword classifier to retrieve bug-revealing keywords. The experimental results have demonstrated the effectiveness of our extracted keywords in facilitating bug localization performance. To understand whether the benefits of golden keywords extend beyond bug localization, we extended our evaluation to another four bug report management tasks, including bug severity prediction, bug priority prediction, bug reopen prediction, and bug field reassignment prediction. The four tasks are generally resolved as a classification problem that mainly involves the content analysis of bug reports.

We followed the strategy of Chen et al. (2024) to label each bug report. Then, we randomly selected 80% of the bug reports as the training dataset, while the remaining 20% were reserved as the test dataset. Table 39 shows the number of instances belonging to different classes in the four tasks. From the table, we could observe a notable class imbalance problem across the training datasets (i.e., some classes have many more instances than others). Such imbalance may make the built model biased towards the majority classes during prediction. Toward this, we designed a data augmentation strategy centered on synonym replacement to fix the class imbalance problem, which proved to help achieve better classification performance than purely using traditional balancing strategies, e.g., random under or over-sampling, in our preliminary experiments.

Table 39 Instance Counts per Class in the Four Bug Report Management Tasks

Dataset	Bug Severity		Bug Priority			Bug Reopen		Bug Field Reassignment	
	Severe	Non- Severe	High	Medium	Low	Reopen	Non- Reopen	Reassign	Non-Reassign
Training	3,029	2,289	2,283	15,638	226	1,473	16,681	6,180	11,968
Test	758	572	571	3,910	57	367	4,171	1,545	2,992

Specifically, we generated additional instances by randomly selecting existing instances from the minority class and replacing 50% of their tokens with synonyms. This synonym replacement was facilitated using the BERT model fine-tuned on the original bug reports, with stopwords and standard keywords excluded from the replacement process. The augmentation process continues until the number of minority class instances triples its original size. If the expanded minority class still has fewer instances than the majority class, random

undersampling (RUS) is applied to the majority class to achieve balance. Conversely, if the minority class surpasses the majority class in size after augmentation, additional instances for the majority class instances will be generated through synonym replacements until both classes are equal in size. Table 40 shows the instance distribution after balancing. Note that we only performed class balancing on the training dataset while keeping the testing dataset imbalanced to mimic real-world scenarios.

Table 40 Instance Counts per Class in the Four Bug Report Management Tasks after Performing Class Balancing based on the Data Augmentation Strategy

Dataset	Bug Severity		Bug Priority			Bug Reopen		Bug Field Reassignment	
	Severe	Non- Severe	High	Medium	Low	Reopen	Non- Reopen	Reassign	Non-Reassign
Training	6,867	6,867	678	678	678	4,419	4,419	18,540	18,540
Test	758	572	571	3,910	57	367	4,171	1,545	2,992

After the class labeling and balancing process, we applied the vector space model with tf-idf term weighting to represent the original content of bug reports and their corresponding golden keywords. Then, we trained random forest models based on the constructed datasets to perform predictions separately. Table 41 shows the classification performance for four studied tasks related to using the original bug reports and golden keywords. From the table, we can find that, when compared to models trained on the original contents of bug reports, using golden keywords did not lead to better classification performance but a slight performance decline across the four studied tasks beyond bug localization. This also provides some support for our claim that providing supervised locating guidance is necessary for facilitating bug localization; our construction and further leveraging of golden keywords is such an attempt in this direction.

Table 41 The Classification Performance Comparison Between Using Original Contents of Bug Reports and Their Corresponding Golden Keywords

Task	Item	F1 score	Precision	Recall
Bug Severity Prediction	Original Content	0.67	0.74	0.62
	Golden Keywords	0.65	0.71	0.60
Bug Priority Prediction	Original Content	0.56	0.56	0.56
	Golden Keywords	0.53	0.53	0.53
Bug Reopen Prediction	Original Content	0.74	0.79	0.70
	Golden Keywords	0.71	0.76	0.67
Bug Field Reassignment Prediction	Original Content	0.65	0.78	0.56
	Golden Keywords	0.62	0.72	0.55

A.7 Replacing Word2Vec with BERT During Keyword Classifier Building

In the keyword-classifier building step, we have several instance features involving semantic similarity calculation based on semantic vectors generated by the typical Word2Vec that was trained on bug reports. To explore whether the use of more advanced embedding techniques would lead to better keyword classification performance, we tried to replace Word2Vec with BERT during instance feature extraction. Like Word2Vec, we also fine-tuned BERT with bug reports (using its built-in masked language modeling task). Table 42 presents the corresponding keyword classification performance for using Word2Vec and BERT, respectively. From the table, we can find that the F1 scores are almost the same. In other words, replacing Word2Vec with BERT does not lead to a substantial improvement in the classification performance of our keyword classifier. This may be because the number of features computed using the word embedding technique during feature extraction is relatively small; these features themselves did not play a dominant role in keyword classification (which could also, to some extent, be revealed by our feature importance analysis in Section 5.2).

Table 42 The Classification Performance of Keyword Classifiers When Using Word2Vec and BERT to Calculate Relevant Instance Features

Project	Method	F1 score	Precision	Recall
AspectJ	Word2Vec	0.45	0.31	0.71
	BERT	0.45	0.46	0.45
Birt	Word2Vec	0.42	0.38	0.47
	BERT	0.43	0.41	0.45
Eclipse.Platform.UI	Word2Vec	0.43	0.34	0.58
	BERT	0.42	0.32	0.62
JDT	Word2Vec	0.45	0.34	0.65
	BERT	0.45	0.38	0.56
SWT	Word2Vec	0.46	0.35	0.67
	BERT	0.45	0.37	0.58
Tomcat	Word2Vec	0.42	0.35	0.53
	BERT	0.43	0.38	0.5

Author Contributions Biyu Cai: Writing – review & editing, Writing – original draft, Methodology, Investigation, Data Curation, Software. Weiqin Zou: Writing – review & editing, Polish. Qianshuang Meng: Methodology. Hui Xu: Methdology. Jingxuan Zhang: Writing – review & editing, Polish.

Funding This work is supported by the National Natural Science Foundation of China (No.62002161, 62272225), partly supported by Key Laboratory of Safety-Critical Software (Nanjing University of Aeronautics and Astronautics), Ministry of Industry and Information Technology (Grant No. 56XCA2002605), the Open Project Foundation of State Key Lab. for Novel Software Technology, Nanjing University (Grant No. KFKT2024B35), and Collaborative Innovation Center of Novel Software Technology and Industrialization.

Data Availibility The datasets and code scripts for replication are available in the KBL repository(https://github.com/Caiby0927/KBL).

Declarations

Conflicts of interest The authors declare no conflict of interest.

Ethical Approval Ethical approval not applicable.

Informed Consent Informed consent not applicable.

Clinical Trial Number Clinical Trial Number not applicable.

References

Bettenburg N, Just S, Schröter A, Weiss C, Premraj R, Zimmermann T (2008) What makes a good bug report? In: Proceedings of the 16th ACM SIGSOFT International Symposium on Foundations of software engineering, pp 308–318

Blei DM, Ng AY, Jordan MI (2003) Latent dirichlet allocation. J Mach Learn Res 3(Jan):993–1022

Carpineto C, Romano G (2012) A survey of automatic query expansion in information retrieval. Acm Comput Surv (CSUR) 44(1):1–50

Chaparro O, Marcus A (2016) On the reduction of verbose queries in text retrieval based software maintenance. In: Proceedings of the 38th International Conference on Software Engineering Companion, Association for Computing Machinery, New York, NY, USA, ICSE '16, p 716–718, https://doi.org/10.1145/2889160.2892647

Chaparro O, Florez JM, Marcus A (2017) Using observed behavior to reformulate queries during text retrieval-based bug localization. In: 2017 IEEE International Conference on Software Maintenance and Evolution (ICSME), pp 376–387, https://doi.org/10.1109/ICSME.2017.100

Chaparro O, Florez JM, Marcus A (2019) Using bug descriptions to reformulate queries during text-retrieval-based bug localization. Empir Softw Eng 24:2947–3007

Chen B, Zou W, Cai B, Meng Q, Liu W, Li P, Chen L (2024) An empirical study on the potential of word embedding techniques in bug report management tasks. Empir Softw Eng 29(5):122

Church KW (2017) Word2vec. Natural Language Eng 23(1):155–162

Dallmeier V, Zimmermann T (2007) Extraction of bug localization benchmarks from history. In: Proceedings of the 22nd IEEE/ACM international conference on automated software engineering, pp 433–436

Deerwester S, Dumais ST, Furnas GW, Landauer TK, Harshman R (1990) Indexing by latent semantic analysis. J American Soc Inf Sci 41(6):391–407

Florez JM, Chaparro O, Treude C, Marcus A (2021) Combining query reduction and expansion for text-retrieval-based bug localization. In: 2021 IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER), pp 166–176, https://doi.org/10.1109/SANER50967.2021.00024

Gay G, Haiduc S, Marcus A, Menzies T (2009) On the use of relevance feedback in ir-based concept location. In: 2009 IEEE International Conference on Software Maintenance, pp 351–360, https://doi.org/10.1109/ICSM.2009.5306315

Haiduc S, Bavota G, Marcus A, Oliveto R, De Lucia A, Menzies T (2013) Automatic query reformulations for text retrieval in software engineering. In: 2013 35th International Conference on Software Engineering (ICSE), pp 842–851. https://doi.org/10.1109/ICSE.2013.6606630

Huo X, Thung F, Li M, Lo D, Shi ST (2019) Deep transfer bug localization. IEEE Trans Softw Eng 47(7):1368–1380

Jones JA, Harrold MJ (2005) Empirical evaluation of the tarantula automatic fault-localization technique. In: Proceedings of the 20th IEEE/ACM International Conference on Automated Software Engineering, Association for Computing Machinery, New York, NY, USA, ASE '05, p 273–282. https://doi.org/10.1145/1101908.1101949

Kevic K, Fritz T (2014) Automatic search term identification for change tasks. In: Companion Proceedings of the 36th International Conference on Software Engineering, Association for Computing Machinery, New York, NY, USA, ICSE Companion 2014, p 468–471. https://doi.org/10.1145/2591062.2591117

Kim D, Tao Y, Kim S, Zeller A (2013) Where should we fix this bug? a two-phase recommendation model. IEEE Trans Softw Eng 39(11):1597–1610

Kim IY, De Weck O (2005) Variable chromosome length genetic algorithm for progressive refinement in topology optimization. Struct Multidiscip Optimiz 29:445–456

Kim M, Lee E (2019a) A novel approach to automatic query reformulation for ir-based bug localization. In: Proceedings of the 34th ACM/SIGAPP Symposium on Applied Computing, Association for Computing Machinery, New York, NY, USA, SAC '19, p 1752–1759. https://doi.org/10.1145/3297280.3297451

Kim M, Lee E (2019b) A novel approach to automatic query reformulation for ir-based bug localization. In: Proceedings of the 34th ACM/SIGAPP Symposium on Applied Computing, Association for Computing Machinery, New York, NY, USA, SAC '19, p 1752–1759, https://doi.org/10.1145/3297280.3297451

Kim M, Kim Y, Lee E (2021) A novel automatic query expansion with word embedding for ir-based bug localization. In: 2021 IEEE 32nd International Symposium on Software Reliability Engineering (ISSRE), IEEE, pp 276–287

Kochhar PS, Tian Y, Lo D (2014) Potential biases in bug localization: Do they matter? In: Proceedings of the 29th ACM/IEEE international conference on Automated software engineering, pp 803–814

Kong A, Zhao S, Chen H, Li Q, Qin Y, Sun R, Bai X (2023) Promptrank: Unsupervised keyphrase extraction using prompt. arXiv:2305.04490

Koyuncu A, Bissyandé TF, Kim D, Liu K, Klein J, Monperrus M, Traon YL (2019) D &c: A divide-and-conquer approach to ir-based bug localization. CoRR arXiv:1902.02703

Krejcie R (1970) Determining sample size for research activities. Education Psychol Meas

Lam AN, Nguyen AT, Nguyen HA, Nguyen TN (2017) Bug localization with combination of deep learning and information retrieval. In: 2017 IEEE/ACM 25th International Conference on Program Comprehension (ICPC), IEEE, pp 218–229

Laurikkala J (2001) Improving identification of difficult small classes by balancing class distribution. In: Artificial Intelligence in Medicine: 8th Conference on Artificial Intelligence in Medicine in Europe, AIME 2001 Cascais, Portugal, July 1–4, 2001, Proceedings 8, Springer, pp 63–66

Le TDB, Oentaryo RJ, Lo D (2015) Information retrieval and spectrum based bug localization: Better together. In: Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering, pp 579–590

Lee J, Kim D, Bissyandé TF, Jung W, Le Traon Y (2018) Bench4bl: Reproducibility study on the performance of ir-based bug localization. In: Proceedings of the 27th ACM SIGSOFT International Symposium on Software Testing and Analysis, Association for Computing Machinery, New York, NY, USA, ISSTA 2018, p 61–72. https://doi.org/10.1145/3213846.3213856

Li Z, Jiang Z, Chen X, Cao K, Gu Q (2021) Laprob: a label propagation-based software bug localization method. Inf Softw Technol 130:106410

Lukins SK, Kraft NA, Etzkorn LH (2008) Source code retrieval for bug localization using latent dirichlet allocation. In: 2008 15th Working Conference on Reverse Engineering, pp 155–164. https://doi.org/10.1109/WCRE.2008.33

Luo Z, Wang W, Caichun C (2023) Improving bug localization with effective contrastive learning representation. IEEE Access 11:32523–32533. https://doi.org/10.1109/ACCESS.2022.3228802

Macbeth G, Razumiejczyk E, Ledesma RD (2011) Cliff's delta calculator: A non-parametric effect size program for two groups of observations. Universitas Psych 10(2):545–555

Mann HB, Whitney DR (1947) On a test of whether one of two random variables is stochastically larger than the other. The Annal Math Stat pp 50–60

Mihalcea R, Tarau P (2004) Textrank: Bringing order into text. In: Proceedings of the 2004 conference on empirical methods in natural language processing, pp 404–411

Mills C, Parra E, Pantiuchina J, Bavota G, Haiduc S (2020) On the relationship between bug reports and queries for text retrieval-based bug localization. Empir Softw Eng 25:3086–3127

Moreno L, Treadway JJ, Marcus A, Shen W (2014) On the use of stack traces to improve text retrieval-based bug localization. In: 2014 IEEE International Conference on Software Maintenance and Evolution, pp 151–160. https://doi.org/10.1109/ICSME.2014.37

Moreno L, Bavota G, Haiduc S, Di Penta M, Oliveto R, Russo B, Marcus A (2015) Query-based configuration of text retrieval solutions for software engineering tasks. In: Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering, Association for Computing Machinery, New York, NY, USA, ESEC/FSE 2015, p 567–578. https://doi.org/10.1145/2786805.2786859

Rahman MM, Roy CK (2017) Strict: Information retrieval based search term identification for concept location. In: 2017 IEEE 24th International Conference on Software Analysis, Evolution and Reengineering (SANER), pp 79–90. https://doi.org/10.1109/SANER.2017.7884611

Rahman MM, Roy CK (2018a) Improving bug localization with report quality dynamics and query reformulation. In: Proceedings of the 40th International Conference on Software Engineering: Companion Proceeedings, Association for Computing Machinery, New York, NY, USA, ICSE '18, p 348–349. https://doi.org/10.1145/3183440.3195003

Rahman MM, Roy CK (2018) Improving ir-based bug localization with context-aware query reformulation. Ass Comput Mach, New York, NY, USA, ESEC/FSE 2018:621–632. https://doi.org/10.1145/3236024.3236065

Rahman MM, Khomh F, Yeasmin S, Roy CK (2021) The forgotten role of search queries in ir-based bug localization: an empirical study. Empir Softw Eng 26(6):116

Rao S, Kak A (2011) Retrieval from software libraries for bug localization: A comparative study of generic and composite text models. In: Proceedings of the 8th Working Conference on Mining Software Repositories, Association for Computing Machinery, New York, NY, USA, MSR '11, p 43–52. https://doi.org/10.1145/1985441.1985451

Roldan-Vega M, Mallet G, Hill E, Fails JA (2013) Conquer: A tool for nl-based query refinement and contextualizing code search results. In: 2013 IEEE International Conference on Software Maintenance, IEEE, pp 512–515

Saha RK, Lease M, Khurshid S, Perry DE (2013) Improving bug localization using structured information retrieval. In: 2013 28th IEEE/ACM International Conference on Automated Software Engineering (ASE), pp 345–355. https://doi.org/10.1109/ASE.2013.6693093

Sampson JR (1976) Adaptation in natural and artificial systems (john h. holland)

Seiffert C, Khoshgoftaar TM, Van Hulse J, Napolitano A (2009) Rusboost: A hybrid approach to alleviating class imbalance. IEEE Trans Syst, Man, Cybern-Part A: Syst Humans 40(1):185–197

Shao S, Yu T (2023) Information retrieval-based fault localization for concurrent programs. In: 2023 38th IEEE/ACM International Conference on Automated Software Engineering (ASE), pp 1467–1479. https://doi.org/10.1109/ASE56229.2023.00122

Shi X, Xu G, Shen F, Zhao J (2015) Solving the data imbalance problem of p300 detection via random undersampling bagging svms. In: 2015 International Joint Conference on Neural Networks (IJCNN), pp 1–5. https://doi.org/10.1109/IJCNN.2015.7280834

Sisman B, Kak AC (2012) Incorporating version histories in information retrieval based bug localization. In: 2012 9th IEEE working conference on mining software repositories (MSR), IEEE, pp 50–59

Sisman B, Kak AC (2013) Assisting code search with automatic query reformulation for bug localization. In: 2013 10th Working Conference on Mining Software Repositories (MSR), pp 309–318. https://doi.org/10.1109/MSR.2013.6624044

Tomek I (1976) Two modifications of cnn. In: IEEE Transactions on Systems Man & Cybernetics, https://doi.org/10.1109/TSMC.1976.4309452

Wang S, Lo D (2014) Version history, similar report, and structure: Putting them together for improved bug localization. In: Proceedings of the 22nd International Conference on Program Comprehension, Association for Computing Machinery, New York, NY, USA, ICPC 2014, p 53–63. https://doi.org/10.1145/2597008.2597148

Wang S, Lo D (2016) Amalgam+: Composing rich information sources for accurate bug localization. J Softw: Evol Process 28(10):921–942

Wen M, Wu R, Cheung SC (2016) Locus: Locating bugs from software changes. In: Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering, pp 262–273

Wong CP, Xiong Y, Zhang H, Hao D, Zhang L, Mei H (2014) Boosting bug-report-oriented fault localization with segmentation and stack-trace analysis. In: 2014 IEEE international conference on software maintenance and evolution, IEEE, pp 181–190

Xiao X, Xiao R, Li Q, Lv J, Cui S, Liu Q (2023) Bugradar: Bug localization by knowledge graph link prediction. Inf Softw Technol p 107274

Xiao Y, Keung J, Mi Q, Bennin KE (2017) Improving bug localization with an enhanced convolutional neural network. In: 2017 24th Asia-Pacific Software Engineering Conference (APSEC), IEEE, pp 338–347

Yan M, Xia X, Fan Y, Hassan AE, Lo D, Li S (2020) Just-in-time defect identification and localization: A two-phase framework. IEEE Trans Softw Eng 48(1):82–101

Ye X, Bunescu R, Liu C (2014) Learning to rank relevant files for bug reports using domain knowledge. In: Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering, Association for Computing Machinery, New York, NY, USA, FSE 2014, p 689–699. https://doi.org/10.1145/2635868.2635874

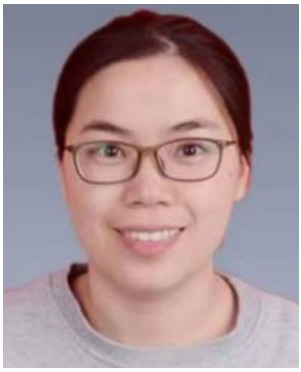
Ye X, Shen H, Ma X, Bunescu R, Liu C (2016) From word embeddings to document similarities for improved information retrieval in software engineering. In: Proceedings of the 38th international conference on software engineering, pp 404–415

Yoo S, Xie X, Kuo FC, Chen TY, Harman M (2017) Human competitiveness of genetic programming in spectrum-based fault localisation: Theoretical and empirical analysis. ACM Trans Softw Eng Method 26(1). https://doi.org/10.1145/3078840

Youm KC, Ahn J, Kim J, Lee E (2015) Bug localization based on code change histories and bug reports. In: 2015 Asia-Pacific Software Engineering Conference (APSEC), IEEE, pp 190–197

Zhou J, Zhang H, Lo D (2012) Where should the bugs be fixed? more accurate information retrieval-based bug localization based on bug reports. In: 2012 34th International conference on software engineering (ICSE), IEEE, pp 14–24

Biyu Cai She received her master's degree from Nanjing University of Aeronautics and Astronautics. Her main research interests include bug localization.



Weiqin Zou She is an associate professor in the College of Computer Science and Technology at Nanjing University of Aeronautics and Astronautics. She received her Ph.D degree at the Software Institute, Nanjing University, China, advised by Prof. Baowen Xu and Prof. Zhenyu Chen in 2019. Her research focuses on mining software repositories (e.g., bug reports, GitHub data) to uncover interesting and actionable information to help improve software quality and developer productivity.

Qianshuang Meng He received his postgraduate degree from Nanjing University of Aeronautics and Astronautics, China. His research interests include bug report analysis and natural language processing.

Hui Xu She is a graduate student at Nanjing University of Aeronautics and Astronautics, China. Her research interest includes natural language processing and bug localization.

Jingxuan Zhang He is an associate professor in the College of Computer Science and Technology, Nanjing University of Aeronautics and Astronautics, China. Zhang received the Ph.D degree in software engineering from the Dalian University of Technology, China. His current research interests include mining software repositories and software data analysis.

Authors and Affiliations

Biyu Cai[1] · Weiqin Zou[1] · Qianshuang Meng[1] · Hui Xu[1] · Jingxuan Zhang[1]

✉ Weiqin Zou
weiqin@nuaa.edu.cn

Biyu Cai
caibiyu@nuaa.edu.cn

Qianshuang Meng
qs_meng@nuaa.edu.cn

Hui Xu
lyraxv@nuaa.edu.cn

Jingxuan Zhang
jxzhang@nuaa.edu.cn

[1] Department of Computer Science and Technology, Nanjing University of Aeronautics and Astronautics, Nanjing, China