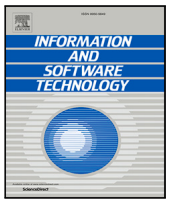
Information and Software Technology 189 (2026) 107943

Contents lists available at ScienceDirect

Information and Software Technology

journal homepage: www.elsevier.com/locate/infsof



Keys4BR: Key sentences-based model fine-tuning for better semantic representation of bug reports

Mengjiao Wang[1], Biyu Cai[1], Weiqin Zou *, Jingxuan Zhang

College of Computer Science and Technology, Nanjing University of Aeronautics and Astronautics, Nanjing, China

ARTICLE INFO

Keywords:
Bug report management
Semantic representation
Large language model fine-tuning
Key information

ABSTRACT

Context: Large language models have been increasingly applied to semantic representation of bug reports due to their deep understanding of natural language. Fine-tuning large language models using bug report text is a common practice to enable models to learn domain-specific knowledge. However, the varying quality of the bug reports can introduce noise, leading to poor performance in downstream tasks.

Objective: To improve the quality of semantic representation for bug reports, we propose Keys4BR, a key sentences-based model fine-tuning for better semantic representation of bug reports.

Method: Specifically, we use keywords that help accurately localize bugs as anchors, designing and applying a key sentences selection strategy to choose portions of the text containing these keywords as the key information. Then we select the lightweight fine-tuning approach to fine-tune the large language model.

Results: Experiments on bug reports from five open-source projects demonstrate that Keys4BR significantly improves the performance of four downstream tasks. The results indicate that Keys4BR achieves superior semantic representation of bug reports compared to the VSM model, the model pre-trained on the general corpus, and the model fine-tuned on original bug reports, with an average F1 score improvement of 9%, 9%, and 6%, respectively. Additionally, we further validate the effectiveness of the key sentences selection and fine-tuning strategies.

Conclusion: Keys4BR can effectively extract key semantic information from bug reports, thereby enhancing the representation capability and generalization performance of large language models in bug report management tasks.

1. Introduction

Bug reports are crucial information carriers that enable developers to comprehend, locate, and fix software bugs [1]. Influenced by factors such as the increasing complexity of software systems and the evolving nature of bug feedback mechanisms (e.g., transitioning from internal team submissions to publicly accepting bug reports from any user), software projects are facing new challenges. Currently, software projects, especially mainstream open-source projects, typically need to process a large number of bug reports under limited human and material resources. For instance, the Mozilla project received up to 10,000 software bug reports within just two months [1]. Under manual processing, projects like ArgoUML and PostgreSQL have a median bug report closure time of nearly 200 days [2]. To reduce the burden on developers in managing bug reports and improve processing efficiency, researchers have proposed various automated bug report management techniques, such as severity prediction, priority prediction, and bug localization [3]. Among these, the accurate understanding and representation of the problem semantics in bug reports are pivotal to the success of such technologies.

Traditional methods for semantic representation or extraction of bug reports typically include the Vector Space Model (VSM) and topic models [4–6], which are insufficient in capturing the contextual semantics of bug reports. Given the deep semantic understanding capabilities demonstrated by large language models (LLMs) in natural language text [7], some researchers have begun to incorporate large models such as BERT [8] into the process of semantic representation of bug reports. By enhancing the semantic representation of bug reports, the performance of downstream bug report management techniques can be improved [9,10]. However, since large models are usually trained on massive general corpora like Wikipedia, they may not adequately capture knowledge specific to the software engineering domain [3]. Therefore, before directly using large models to represent the semantics of bug report text, a common strategy to enhance the performance

* Corresponding author.
E-mail addresses: wangmj@nuaa.edu.cn (M. Wang), caibiyu@nuaa.edu.cn (B. Cai), weiqin@nuaa.edu.cn (W. Zou), jxzhang@nuaa.edu.cn (J. Zhang).
[1] Mengjiao Wang and Biyu Cai are both first authors, and contributed equally to the work.

https://doi.org/10.1016/j.infsof.2025.107943
Received 9 July 2025; Received in revised form 20 October 2025; Accepted 20 October 2025
Available online 23 October 2025

of these models is to fine-tune them with domain-specific data from bug reports. The quality of the domain data used for fine-tuning significantly impacts the final effectiveness of the fine-tuned model [11]. Existing research has found that the quality of bug reports is often variable, and using their full text content for model fine-tuning does not always improve the performance of downstream tasks [3,12]. How to denoise bug report data and utilize the enhanced quality bug report data to fine-tune large models, thereby improving downstream tasks, is an important issue worthy of research.

In response, this paper proposes Keys4BR, a key sentences-based model fine-tuning for better semantic representation of bug reports. The core idea of Keys4BR is to denoise the original bug reports by extracting key information that truly reflects the semantics of software issues, and then use this key information to fine-tune large models, thereby improving the performance of bug report management techniques. The main challenges to be addressed in this research are how to extract this key information and how to fine-tune the large models. Regarding the extraction of key information, this paper uses keywords that help accurately localize bugs as anchors, designing and applying a key sentences selection strategy to choose portions of the text containing these keywords as the key information. The acquisition of these keywords is primarily achieved through the combined use of the genetic algorithm and keyword optimization heuristic rules. Once the key information is obtained, it is input into the large model for fine-tuning. To reduce the cost of fine-tuning, a lightweight fine-tuning strategy is employed (i.e., only a small number of parameters are fine-tuned). The fine-tuned large model is then used to semantically represent the content of the original bug reports for downstream tasks. And the bug report management model is established based on this semantic representation to handle bug reports.

Under the constraint of limited resources, to handle bug reports as efficiently as possible, researchers have developed a range of bug report management (BRM) techniques [13]. These techniques cover multiple aspects, including bug localization, bug severity prediction, bug priority prediction, as well as bug fix time prediction. This section focuses on the bug severity prediction, bug priority prediction, bug reopened prediction, and bug field reassignment prediction that we have involved in our research, which have also been highly recognized by software practitioners [13,14]. These techniques provide us with a purer perspective to evaluate the effectiveness of the Keys4BR in representing the semantics of bug reports. The core advantage lies in the fact that these techniques are mainly based on the analysis of the summaries and descriptions of bug reports, thereby avoiding the potential interference from interactions between bug reports and other software artifacts, as well as the impact of imprecise processing steps such as bug-code linking that are common in other bug report management techniques. During the evaluation phase, we extract the semantic representation vectors of bug reports using the fine-tuned large model and input these vectors into the classifier. Subsequently, we use the trained classifier to classify the test data of bug reports and evaluate its performance.

The experimental dataset consists of bug report data from five open-source projects, totaling 21,691 software bug reports, of which 4,338 reports are designated as the test set for Keys4BR, with the remaining bug reports used for fine-tuning the large model and constructing the classification model. The experimental results demonstrate that the key information extraction strategy proposed in this study is indeed effective. The large model fine-tuned based on bug key information significantly improves downstream bug report management tasks compared to two semantic representation baseline techniques (i.e., fine-tuning the large model using original bug reports and using the traditional Vector Space Model for semantic representation). In the four downstream tasks, the F1 scores show substantial increases over the baseline, with an average absolute improvement of 0.09 and 0.08. To better understand the technical performance of Keys4BR, this paper also compared the effectiveness of different key sentences selection strategies and model fine-tuning strategies.

In summary, this study makes the following contributions:

(1) This study proposes a bug report semantic representation method, Keys4BR, based on fine-tuning with key bug information. The method suggests fine-tuning a large model using denoised key semantic information from bug reports, thereby enhancing the semantic representation of bug reports and further improving downstream bug report management tasks.
(2) This study designs a key information extraction scheme that combines keyword extraction and key sentences selection. This scheme effectively aids in extracting key text information that truly embodies the semantics of software issues, providing a valuable technical reference for denoising the content of bug reports.
(3) This study evaluates the effectiveness of Keys4BR through four downstream bug report management tasks across five open-source projects. The experimental results confirm that Keys4BR technology brings performance improvements to downstream tasks compared to three baseline techniques, and also validate the importance of the key sentences selection strategy and the fine-tuning strategy.

2. Background & related work

In this section, we first introduce the information fields contained in a bug report. Next, we summarize the main semantic representation methods employed in BRM tasks. Finally, we provide a brief overview of the classification models used in four typical downstream BRM classification tasks for evaluating Keys4BR. Details are as follows.

Bug reports play a crucial role in bug analysis by providing detailed information about bugs encountered during software development and use. A bug report typically includes fields such as bug ID, status, importance, summary description, and detailed description.[2] The bug ID is a unique number used to identify a bug in the software system. The summary description is usually a brief sentence that describes the bug, while the detailed description provides comprehensive information about it. This detailed information often includes observed behavior, expected behavior, steps to reproduce the bug, etc. The status field indicates the current status of the bug report,[3] such as ‘Closed’, ‘Fixed’, ‘Resolved’, ‘Reopened’, etc. In some cases, closed bug reports may get reopened, and some may get reassigned to different products or components during their lifecycles (from reporting to closure). And the importance field is divided into two parts: priority and severity. Generally speaking, the higher the severity of a bug, the higher its corresponding priority.

To reduce the time developers spend handling bugs, researchers have proposed various automated methods to support bug-report management tasks, such as bug severity/priority prediction and reopened-bug prediction [13]. Most of these techniques involve the semantic representation of bug reports (BRs). Among them, information retrieval models are most widely adopted, with the Vector Space Model (VSM) being the predominant approach, while topic models such as LDA are sometimes also used to capture the semantic information of bug reports [15–17]. The Vector Space Model assigns weights to each word in a document through, e.g., Term Frequency-Inverse Document Frequency (TF-IDF) weighting strategy [18], thereby representing the document as a vector [15]. Topic Models like LDA generally take a document as a mixture of topics, and would represent a document as a probabilistic distribution over latent topics [19].

Both VSM and LDA take documents as a bag of words without taking their contextual semantics into account. To mitigate this limitation, some deep learning models, more specifically, pre-trained models like Word2Vec [20], BERT [8] and its variants (trained on general website

[2] https://bugs.eclipse.org/bugs/show_bug.cgi?id=155972
[3] https://wiki.documentfoundation.org/QA/Bugzilla/Fields/Status/RESOLVED

contents like Wikipedia and Google News), are gradually adopted by researchers to facilitate bug report management tasks [3]. All these pre-trained models would transform a bug report into an N-dimensional semantic vector and are expected to better capture deep semantic information within bug reports. To improve the performance of pre-trained models, domain-specific data can be used to fine-tune the model. Currently, the state-of-the-art pre-trained model, namely RepresentThemAll [21], is trained on a large-scale dataset of bug reports using deep learning networks.

The above-mentioned pretrained models are universal representation approaches that aim to learn general semantic representations that are independent of downstream tasks, thereby providing a unified semantic feature input applicable across multiple BRM tasks. Our Keys4BR also falls into this category, aiming to fine-tune the general pre-trained model RoBERTa using denoised key information from bug reports. Actually, besides these universal representation approaches, there are also task-specific approaches designed to address a single BRM task, which typically optimize task-specific labels by employing sophisticated models like graph convolutional networks to capture task-relevant features. The PPWGCN [22] for priority prediction is such an example. These models are generally tightly coupled to associated BRM tasks. In this study, we focus primarily on universal models and, accordingly, compare Keys4BR mainly against existing approaches of this type.

To evaluate our Keys4BR, we selected four typical BRM classification tasks that mainly involve content analysis of bug reports, namely Bug Severity Prediction, Bug Priority Prediction, Bug Reopen Prediction, and Bug Reassignment Prediction. These tasks generally use the aforementioned semantic representation methods to extract semantic features from bug reports, then apply machine learning algorithms to build corresponding BRM models. As for the machine learning algorithms used, bug-priority prediction has evolved from traditional methods (e.g., NB, decision trees) to neural networks and, more recently, to popular deep learning techniques (e.g., CNN, DNN) [17,23]. For severity prediction, most existing studies still rely on traditional machine learning approaches, though some have begun to explore deep learning–based prediction methods [17,24,25]. For the bug reopen prediction task, existing studies have primarily employed traditional machine learning algorithms such as decision trees (C4.5), Naïve Bayes, and ensemble methods like bagging [26,27]. For the bug reassignment prediction task, leveraging multi-label learning algorithms has been a common approach, with techniques such as ML-KNN and its variant Im-ML.KNN being utilized to effectively handle the inherent class imbalance [28].

3. Methodology

This study proposes Keys4BR, a semantic representation method for bug reports based on fine-tuning with key information. As illustrated in Fig. 1, Keys4BR comprises three modules: (1) key information extraction, model fine-tuning, and model evaluation. For key information extraction module, Keys4BR first takes bug reports and source code files as input, and extracts keywords by combining the genetic algorithm and the keyword optimization rules. Subsequently, three types of key sentences selection strategies are introduced to select key sentences from the bug report text as the key information. In the model fine-tuning module, the extracted key information is used as domain-specific data to fine-tune a LLM. An efficient lightweight fine-tuning strategy is adopted, which adjusts only a small number of parameters in the LLM to achieve a model with superior performance. The model evaluation module applies the fine-tuned LLM to the semantic representation of four downstream bug report management tasks and constructs classification models based on this representation. The performance of these models on these tasks is used to evaluate the effectiveness of Keys4BR. Each module of Keys4BR is detailed below.

3.1. Key information extraction

Key information extraction consists of two main components: the extraction of bug localization keywords and the selection of key sentences. The goal is to effectively extract key information from bug reports that truly reflects the actual problems.

3.1.1. Bug localization keyword extraction

This study combines a genetic algorithm with keyword optimization rules to extract localization keywords from bug reports that can help accurately locate the corresponding bugs. If a set of keywords performs well in the bug localization task, it indicates their strong correlation with bug-related code segments, implying successful capture of the core semantic information. Using these localization keywords as anchors, appropriate bug texts containing localization keywords (detailed in Section 3.1.2) are selected to fine-tune large language models (LLMs), thereby enhancing the performance of downstream bug report management tasks. This module references previous research [29] and mainly includes the following two steps: First, a genetic algorithm is used to obtain an initial set of localization keywords; then, keyword optimization rules are employed to refine this set, resulting in the final set of keywords.

Initial Keywords Extraction. During the initial keyword extraction phase, a genetic algorithm is employed to obtain the initial set of localization keywords. In the evolutionary process, the genetic algorithm generates a set of keywords. These keywords replace the original bug report and are input into the Lucene[4] search engine to retrieve a list of related code files. If the truly buggy code file corresponding to the bug report ranks higher in the list, then these keywords are considered more effective in bug localization. Through this approach, the genetic algorithm attempts to generate a query that places the truly buggy file at the top of the recommendation list. The algorithm iteratively evolves the initial population through selection, crossover, and mutation operations until it reaches the optimal fitness score (E value of 1) or 30,000 generations. Following the execution of the genetic algorithm, each bug report will obtain a set of initial keywords for querying, which can achieve nearly optimal results in the retrieval process.

Keyword Optimization. Despite the good performance of the initial keywords in retrieving buggy code files (that is, the median effectiveness metric value for each project in the bug localization task is 1), there may still be some noise. To ensure that the extracted keyword set is as noise-free as possible while maintaining retrieval performance, we apply keyword optimization rules to remove noise from the keywords. Specifically, if adding a word to the query reduces retrieval effectiveness, that word is likely to be a noise word. In this study, the summary and description of the bug report are considered complete text T. We use text window selection to construct subtexts from consecutive sentences and compare the retrieval effectiveness of subtexts to identify noise words. To ensure that the retrieval performance of the final keywords is not lower than that of the initial keywords, the following steps are taken. First, we compare the E metric values of the best sub-keywords and the initial keywords, and regard the keyword set with better performance as the initial best keywords. Next, we remove all previously identified noise words and evaluate the performance of the keywords after removal. If there is improvement, these keywords will be returned as the best keywords. If not, this indicates that some of the removed noise words may contain pseudo-noise words. In this case, the previously removed noise words are added back to the keyword set one by one to assess the changes in retrieval performance. This process helps to identify any pseudo-noise words that may have been mistakenly removed. The keywords with the best performance during this process will be returned as the best keywords and used as the final keywords to locate.

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

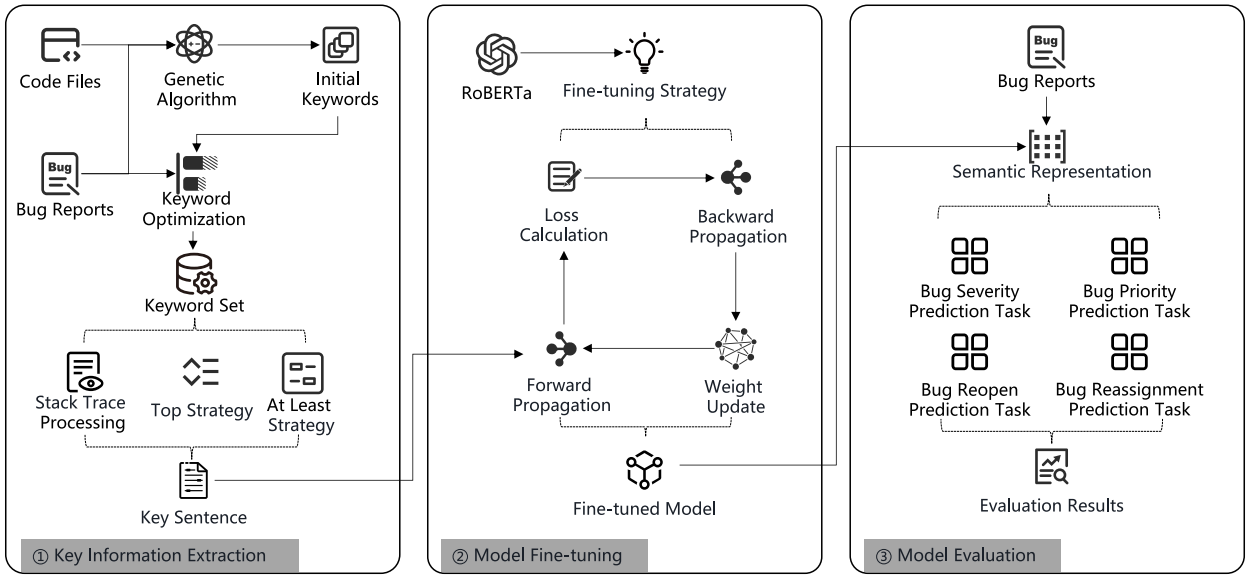


Fig. 1. The overall framework of Keys4BR.

3.1.2. Key sentences selection strategies

After obtaining the optimized keywords, a key sentence selection strategy is required to extract the key information, which is presented in the form of sentences containing these keywords. This section proposes two key information selection strategies: the basic key sentences selection strategy and the stack information processing strategy. Among them, the basic key sentence selection strategy is an indispensable core strategy for extracting key information from bug reports. It includes two specific sub-strategies: The Top Strategy and the At Least Strategy. In contrast, the stack information processing strategy plays a supplementary and expansive role, encompassing three sub-strategies that can enhance the extraction of basic key sentences from bug reports to a certain extent, thereby making the finally extracted key information more accurate and effective.

Basic Key Sentences Selection Strategy. The basic key sentences selection strategy encompasses two distinct sub-strategies: At Least Strategy and The Top Strategy. The following sections elaborate on the fundamental principles of these two strategies and how each of them extracts basic key sentences containing critical information from bug report texts.

(1) **At Least Strategy:** The core objective of the At least Strategy is to select sentences from the bug report that contain at least a specified number n of standard keywords as basic key sentences. Specifically, the first step relies on the semantic keywords in the bug report to count the number of keywords contained in each sentence. Subsequently, based on the predefined hyperparameter n, sentences that meet this criterion are filtered out of all sentences. These selected sentences are then regarded as the basic key sentences of the bug report. This strategy emphasizes the coverage of keywords in the sentences to ensure that the selected sentences comprehensively reflect the core problem description of the bug report. This provides a higher-quality textual basis for subsequent bug localization or semantic representation tasks. This strategy sets a threshold n, and when the number of keywords in the sentence reaches this threshold, the sentence will be considered as key information.

(2) **The Top Strategy:** This strategy aims to select the k sentences with the highest number of keywords in the bug report as the basic key sentences. Unlike the At Least Strategy, which sets a lower bound on the number of keywords contained in the sentences, the Top Strategy directly restricts the number of basic key sentences to be selected. Specifically, the Top Strategy also relies on the semantic keywords in the bug report to perform a statistical analysis of the number of keywords in each sentence. Subsequently, the sentences are sorted in descending order based on the number of keywords they contain, and the k sentences with the most keywords are selected according to the predefined hyperparameter k. These sentences are regarded as the basic key sentences under the Top Strategy. Compared to the At Least Strategy, the Top Strategy places greater emphasis on selecting representative sentences, thereby retaining the key information of the bug report within a limited number of sentences.

Stack Trace Processing Strategy: To better handle the common stack information issues in bug reports, three strategies are further proposed that specifically target stack information. These strategies aim to address the challenges of excessive text length and significant format differences of stack information compared to natural language text. They also reduce the complexity for models processing bug report texts and avoid exceeding the maximum text length limit of LLMs. The specific strategies are as follows:

(1) **Direct Removal Strategy:** The core idea of this strategy is to remove all stack information from the bug report directly, treating it as redundant content. Stack information typically contains numerous method details, such as method call chains and file paths. Although these details may be valuable for manual debugging, they tend to increase the complexity and interference of models during semantic understanding and text representation tasks. By eliminating stack information, the text complexity of the bug report can be significantly reduced, allowing the model to focus more effectively on processing the natural language text portion.

(2) **Exception Name Replacement Strategy:** The first line of stack information typically contains the exception or error name encountered during code execution, which is crucial for bug localization and analysis. This strategy extracts the exception or error name from the stack information and replaces the entire stack information with it. In this way, the most critical information is retained while substantially reducing the complexity of the text. On the one hand, it significantly reduces redundant content and prevents the stack information from occupying the processing capacity of large language models; on the other hand, by focusing on the exception or error name, it provides the model with more direct and clear clues for understanding the problem.

(3) **Template Rewriting Strategy:** To express stack information in natural language, this strategy designs a predefined template to rewrite stack information into a description that more closely reflects the natural language style. The template format is: "The [method name] method of [class name] encountered [exception or error name] in [file name]". Specifically, the first step is to extract key information such as class name, method name, file name, and exception name from the stack information. The extracted information is then filled in the corresponding positions in the template according to the type of information. If some fields (such as method name) cannot be successfully extracted, a simplified template can be used, namely "Encountered [exception or error name]". Through template rewriting, the originally obscure and lengthy stack information is transformed into a direct expression of the core problem. This not only makes the bug report more intuitive and understandable, but also provides higher-quality input for subsequent model analysis and task execution.

3.2. Model fine-tuning

During the model fine-tuning process, we employ lightweight fine-tuning techniques to update the parameters of the LLM (i.e., the RoBERTA[5] with 125 million parameters in this study), that is, only updating the weights of some rather than all parameters. Although full-parameter fine-tuning is computationally feasible, we intentionally choose the lightweight parameter-efficient fine-tuning (PEFT) strategy by considering the following aspects. First, bug reports from open-source projects often vary in quality and frequently contain noise [29,30]. While we try to replace original bug reports with key information, residual noise may persist in low-quality reports. Lightweight fine-tuning methods (e.g., LoRA, AdaLoRA) have been demonstrated to enhance robustness against such noise, reducing overfitting and often achieving performance comparable to, or even surpassing, that of full fine-tuning [31–33]. In addition, these methods substantially reduce memory and runtime costs while producing reusable adapters, making them more suitable for practical deployment scenarios that require frequent updates or migration.

Keys4BR primarily employs the AdaLoRA method for efficient fine-tuning [32]. To validate the effectiveness of AdaLoRA, we also compare it with two other common lightweight fine-tuning approaches during the experimental process: the bottom-frozen fine-tuning and the LoRA [31]. The bottom-frozen fine-tuning involves freezing the parameters of the model's bottom layers, and allowing only the top three layers' parameters to be adjusted, thereby reducing computational load while maintaining the model's basic semantic representation capabilities. The LoRA can achieve efficient parameter updates by fixing most parameters and introducing low-rank matrices to adjust a small number of model weights, thereby reducing training time and memory consumption while ensuring improved model performance. AdaLoRA is a variant of LoRA, which can adaptively adjust the rank of the LoRA matrix, flexibly allocating resources for parameter updates according to the specific needs of the task, balancing model accuracy and computational efficiency.

After obtaining the key information from the bug report through the key information extraction module, it is used to fine-tune the LLM using AdaLoRA. The large language model selected for fine-tuning in this paper is RoBERTa [34], a model pre-trained on large-scale general corpora like Wikipedia and book corpora. We selected RoBERTa as our backbone model mainly because: (1) As an encoder-only architecture, RoBERTa is well-suited for natural language understanding tasks [35], enabling it to effectively capture the semantics of textual key sentences in bug reports; (2) Adopting RoBERTa could help us avoid the additional decoder overhead present in encoder–decoder models like T5-base [36], leading to reduced memory and training costs [37]; (3) Compared to other encoder-only models such as DeBERTa [38,39], RoBERTa offers a more favorable balance between performance and training stability, particularly in resource-constrained scenarios [40]; and (4) RoBERTa is widely supported by modern parameter-efficient fine-tuning tools including AdaLoRA, which facilitates stable and efficient adaptation. The task used for fine-tuning the model is the Masked Language Modeling (MLM) task. MLM is a self-supervised learning task that involves randomly masking some words in the input text and requiring the model to predict the masked words based on the context, thereby improving the model's ability to understand context-rich sentences. After fine-tuning, the model is able to capture fine-grained key information, thereby better representing the semantic information of the bug report.

When the text of a bug report is input into the fine-tuned model, the model generates a 768-dimensional embedding vector, which condenses the deep semantic information of the entire bug report. This vector representation takes into account the lexical, syntactic, and semantic information of the bug report, providing a more comprehensive understanding of the text. This vector can serve as a semantic foundation for downstream bug report management tasks (e.g., severity and priority prediction), thereby helping to enhance the model's performance on these downstream tasks.

3.3. Model evaluation

To evaluate the effectiveness of the fine-tuned model, we apply it to four downstream tasks in the field of bug report management. Firstly, the fine-tuned model is used to generate semantic vectors for bug reports. These vectors serve as features of the bug reports and are input into the classifier. The performance of the classification will reflect the effectiveness of the fine-tuned model. The four downstream tasks selected to evaluate our Keys4BR are bug severity prediction, bug priority prediction, bug reopen prediction, and bug reassignment prediction. These tasks are typical bug report management tasks, involving mainly the analysis of a bug report's content. They are generally taken as classification tasks by the academic community. After obtaining the semantic features of a bug report, we further need to determine its labels in individual tasks, so that we can build classifiers based on labeled datasets. During the labeling process, we mainly refer to the available raw metadata recorded in the corresponding information items (e.g., "P1" for Priority item, "Block" for Severity item) of each bug report in the issue tracking system (e.g., Bugzilla[6]). Based on the collected metadata, we follow the mainstream practice in prior studies to label each bug report correspondingly. Labeling details for the four tasks are as follows.

(1) **Bug Severity Prediction Task** aims to predict the severity level of a bug report, indicating the seriousness of the corresponding bug issue. According to the settings of bug tracking systems (e.g., Eclipse and Apache), the severity field of a bug report can include seven values: Blocked, Critical, Major, Normal, Minor, Trivial, and Enhancement. Following previous research [5,41,42], we categorize bug reports into two classes based on the value of severity field: *severe* and *non-severe*. Specifically, bug reports with severity values of Blocked, Critical, and Major are classified as *severe*, while those with severity values of Minor, Trivial, and Enhancement are classified as *non-severe*.

(2) **Bug Priority Prediction Task** aims to automate the prediction of bug report priority, addressing the time-consuming and error-prone issues inherent in manual assignment processes. The priority field of a bug report contains five levels, namely P1, P2,

[5] https://huggingface.co/FacebookAI/roberta-base

[6] https://www.bugzilla.org

P3, P4, and P5. Specifically, P1 represents the highest priority, while P5 represents the lowest priority. Referencing previous research [43,44], we divide these five levels into three categories: *high* (including P1 and P2 levels), *medium* (including P3 level), and *low* (including P4 and P5 levels) priority.

(3) **Bug Reopen Prediction Task** refers to those bugs that have been marked as closed by developers but are later reopened. For each resolved bug report, by checking the content of its history item, if its historical status has ever been changed to reopened, it is classified as *reopen*; otherwise, it is classified as *non-reopen*.

(4) **Bug Reassignment Prediction Task** aims to predict in advance which fields in a bug report might be reassigned. For each resolved bug report, we first check whether the values of its status, component, product, priority, severity, operating system, and version fields have changed after initial assignment [3]. If any of the above fields have changed, the bug report is classified as *reassign*; otherwise, it is classified as *non-reassign*.

4. Experimental setup

This section firstly describes how to construct the dataset for the experiments, then introduces the metrics used to evaluate the performance of Keys4BR, and finally proposes four research questions that this study aims to address.

4.1. Experimental dataset

In the experiments of this study, it is necessary to construct a dataset of key information for model fine-tuning. Subsequently, datasets corresponding to four downstream bug report management tasks need to be built to evaluate the effectiveness of the methods proposed in this paper. We mainly leverage the dataset kindly shared by Ye et al. [45] and widely used by the academic community [46–49]. This dataset comprises over 20,000 bug reports from five representative open-source projects (such as AspectJ and JDT) of the Eclipse Foundation, each associated with its corresponding buggy code files. Within the dataset, 80% of the bug reports are randomly selected for the fine-tuning of the LLM and the construction of the training set for downstream software tasks. The remaining 20% of the bug reports are used to test the effectiveness of the techniques proposed in this paper. In this study, we chose to pool all bug reports before the data split mainly by considering two aspects. First, it allows us to evaluate the cross-project generalization capability [6,12] of Keys4BR, which is crucial for projects with limited or no historical bug reports for fine-tuning. Second, pooling to some degree helps mitigate data sparsity and imbalance problems, as shown in Table 1, e.g., AspectJ with only 593 reports, where project-wise splits would provide insufficient samples for stable training in the high-dimensional 768-d RoBERTa feature space [6]; a larger, more balanced dataset enables fairer and more stable training and comparison.

All bug reports require preprocessing before they can be used for model fine-tuning, classifier training and testing. The preprocessing steps are as follows: First, we will extract the summary and the description from each resolved bug report. Then, the tokenization will be performed on the text, where the camel-case and dot-separated words will be split into individual words. Non-alphabetic characters are removed from the text, and all words are converted to their lowercase form. In addition, a standard English stop words list [42] is used to eliminate words that frequently appear but contribute little to understanding the text. Finally, the Porter tool will be used to convert words to their stem form. After preprocessing, the bug reports can serve as the foundation for the construction of the key information dataset and the downstream task datasets. The specifics are as follows.

The bug report dataset provided in literature [45] we used includes not only bug reports but also the truly buggy code files related to fixing these bugs. This provides a data foundation for the application of the genetic algorithm and keyword optimization rules to extract and optimize keywords in this paper. By analyzing the buggy code files for bug fixing, it is found that some bugs were fixed solely by adding new code files, without modifying existing code files. In such cases, we removed these bug reports (i.e., a total of 310) from the dataset, as they could not be used to locate the buggy code files to be fixed through bug localization tools. Ultimately, 21,691 bug reports were obtained, and their data distribution is shown in Table 1.

Table 1
The number of bug reports for five open source projects.

Project name	Number of bug reports
AspectJ	593
Birt	4,178
Eclipse.Platform.UI	6,495
JDT	6,274
SWT	4,151

For the randomly selected 80% of the bug reports, key information needs to be extracted for fine-tuning the LLM. The detailed steps are as follows: First, the source code files need to be preprocessed, following the same preprocessing steps as for the bug reports. And the preprocessed bug reports and source code files will be input into the genetic algorithm and keyword optimization rules to obtain the bug report localization keywords. After applying the key sentences selection strategies (detailed in Section 3.1.2) with the extracted localization keywords, the key information corresponding to each bug report can be obtained, and the key information constitutes the domain dataset for fine-tuning the LLM. During the extraction of localization keywords, referring to the settings of existing research [50], the parameters of the genetic algorithm in this paper are set as follows: Population size: 500, Crossover probability: 0.9, Mutation probability: $1/n$ (where n is the number of terms in the bug report), Maximum number of evolutionary generations: 30,000.

4.1.1. Construction of datasets for downstream tasks

For all bug reports from the five open-source projects, labels need to be assigned to the bug reports according to different downstream tasks (the labeling method is detailed in Section 3.3). The label categories and instance numbers corresponding to each downstream task are shown in Table 2. As can be seen from Table 2, there is a significant issue of data imbalance in some of the datasets for downstream tasks. Training directly on this imbalanced dataset could lead to severe biases in classifier performance. To address this, we attempt to employ data augmentation techniques to generate new training instances through synonym replacement, aiming to achieve a balance in the proportion of minority and majority class instances.

Specifically, for each label category of the downstream tasks, 80% of the bug reports are randomly selected as the training set, and the remaining 20% are used as the test set. For the minority class in these 80% training instances, new instances are generated by randomly selecting instances and replacing 50% of the words with synonyms. The synonym replacement operation is carried out using a RoBERTa model fine-tuned on the original bug reports, ensuring that the selected words are neither stopwords nor standard keywords. This instance construction process is repeated until the minority class instances are expanded to three times their original size. If the instance number of the expanded minority class is still fewer than the majority class, the random undersampling (RUS) is performed on the majority class to equalize the numbers. Conversely, if the minority class has more instances after expansion, the majority class is subjected to the same instance expansion process until its size matches that of the expanded minority class. The final experimental dataset, after imbalance processing, the final instance distribution for each label category is as shown in Table 3. It is important to note that in this study, the imbalance processing is only applied to the training set, while the test set retains its original

Table 2
The sample distribution in the original dataset.

Dataset type	Bug severity prediction		Bug priority prediction			Bug reopen prediction		Bug reassignment prediction	
	Severe	Non-severe	High	Medium	Low	Reopen	Non-reopen	Reassign	Non-reassign
Training Set	2,898	2,190	2,180	14,954	218	1,402	15,950	5,910	11,443
Test Set	725	547	545	3,739	55	351	3,988	1,477	2,861
Full Dataset	3,623	2,737	2,725	18,693	273	1,753	19,938	7,387	14,304

Table 3
The sample distribution in the balanced dataset.

Dataset type	Bug severity prediction		Bug priority prediction			Bug reopen prediction		Bug reassignment prediction	
	Severe	Non-severe	High	Medium	Low	Reopen	Non-reopen	Reassign	Non-reassign
Training Set	6,570	6,570	654	654	654	4,206	4,206	17,730	17,730
Test Set	725	547	545	3,739	55	351	3,988	1,477	2,861

imbalanced state, so that we can truly reflect the performance of each downstream classification model in real-world application scenarios. The finally constructed training and testing datasets would be used in all subsequent analyses about the overall performance of Keys4BR and the ablation experiments over the impact of individual Keys4BR components.

4.2. Performance metrics

The following metrics are used to evaluate the performance of Keys4BR.

(1) **Precision:** Precision measures the proportion of true positive results among all positive results predicted by the classifier.
(2) **Recall:** Recall measures the proportion of true positive results among all actual positive results.
(3) **F1 Score:** F1 Score is the harmonic mean of precision and recall, providing a balance between the two.

These metrics have been widely applied in previous research [43, 44] for model performance assessment.

4.3. Research questions

This study aims to deeply understand the overall performance of Keys4BR and the role of its key technical components by addressing the following three research questions:

(1) RQ1: How does Keys4BR perform on the four downstream tasks?
(2) RQ2: What impact do different key sentences selection strategies have on the performance of Keys4BR?
(3) RQ3: What impact do different lightweight fine-tuning strategies have on the performance of Keys4BR?

5. Experimental results

5.1. RQ1. How does Keys4BR perform on the four downstream tasks?

In this research question, we aim to evaluate the semantic representation effectiveness of Keys4BR. Specifically, we integrate Keys4BR into the classification model and apply the resulting classifier to four downstream tasks. The performance of the model on these downstream tasks serves as a measure of Keys4BR's effectiveness. To comprehensively examine the performance of Keys4BR with different classification models, we select three classification models: the Support Vector Machine (SVM), the classification layer, and the Convolutional Neural Network (CNN). SVM is a commonly used machine learning model that has been shown to achieve good performance in some bug report management tasks [12]. Adding a classification layer on top of the large language model is the most direct and common approach for applying a large language model to specific classification tasks. Meanwhile, CNN is one of the most frequently used classification models in bug report management research.

Additionally, we select three semantic representation baselines for comparison with Keys4BR. The first baseline method (abbreviated as VSM) uses the traditional vector space model to represent the semantics of bug reports, where the vector elements are the weights of words calculated as their TF-IDF values. This semantic representation method is often used in conjunction with machine learning classifiers and has been shown to achieve good representation performance in certain tasks. The second baseline method utilizes the RoBERTa model fine-tuned on the general corpus to represent the bug reports (abbreviated as PT), which is readily available and has undergone pretraining, demonstrating strong performance in text representation. The third baseline method involves fine-tuning the RoBERTa with the original bug report text to represent the semantics of the bug reports (abbreviated as FT). When applying pretrained models to bug report domain tasks, researchers typically fine-tune them on bug report texts to help the model learn domain-specific knowledge. Therefore, we also use FT for comparison with Keys4BR. The semantic feature vectors obtained from all baselines are used as text features to train the classification model, resulting in the corresponding classification model. Note that, for completeness, we have also compared Keys4BR with the state-of-the-art RepresentThemAll (RTA) [21], a large-scale pre-trained model trained on over 200,000 raw bug reports. Since RTA and Keys4BR are trained on datasets of vastly different sizes (200K+ vs. 18K-) and cannot be aligned under identical fine-tuning settings, this comparison is excluded from the main results. The supplementary results show that Keys4BR achieved comparable performance with RTA while using 10×less data, demonstrating its efficiency and robustness under small-data, noisy-report conditions. Detailed results are provided in Appendix.

Table 4 shows the comparison results of the classification performance of Keys4BR with three semantic representation baselines across the four downstream bug report management tasks. From the table, we can find that, among the combinations of Keys4BR with three classifiers, the best F1 score could reach 91% for bug severity prediction (achieved by Keys4BR+CNN), 61% for bug priority prediction (achieved by Keys4BR+SVM), 81% for bug reopen prediction (achieved by Keys4BR+SVM), and 74% for bug reassignment prediction (achieved by Keys4BR+CNN), respectively, with the optimal classifiers varying across different tasks. Specifically, in the bug severity prediction task, the CNN classifier performs the best, with an F1 score that is 12% and 1% higher than that of SVM and the classification layer, respectively. In the bug priority prediction task, the combination of the SVM classifier and Keys4BR achieves the best performance, with an F1 score that is 9% and 14% higher than that of the classification layer and CNN, respectively. In the bug reopen prediction task, the SVM classifier also performs the best, with an F1 score that is 3% higher than that of both the classification layer and CNN. In the bug reassignment prediction task, the CNN performs best, and the classification layer obtains almost

Table 4
The comparison results of Keys4BR and three baseline methods.

Task	Method	SVM			Classification layer			CNN		
		F1 score	Precision	Recall	F1 score	Precision	Recall	F1 Score	Precision	Recall
Bug Severity Prediction	VSM	68%	76%	61%	–	–	–	–	–	–
	PT	66%	74%	60%	85%	85%	86%	90%	89%	90%
	FT	69%	77%	63%	86%	88%	86%	90%	91%	89%
	Keys4BR	79%	69%	92%	90%	89%	90%	**91%**	**90%**	**91%**
Bug Priority Prediction	VSM	55%	55%	55%	–	–	–	–	–	–
	PT	42%	42%	42%	35%	35%	35%	37%	37%	37%
	FT	48%	48%	48%	47%	47%	47%	41%	41%	41%
	Keys4BR	**61%**	**61%**	**61%**	52%	52%	52%	47%	47%	47%
Bug Reopen Prediction	VSM	73%	77%	70%	–	–	–	–	–	–
	PT	68%	72%	64%	70%	66%	75%	72%	68%	77%
	FT	70%	75%	65%	74%	70%	77%	74%	67%	83%
	Keys4BR	**82%**	**79%**	**85%**	79%	76%	82%	79%	81%	78%
Bug Reassignment Prediction	VSM	64%	79%	54%	–	–	–	–	–	–
	PT	65%	61%	70%	70%	81%	62%	72%	79%	65%
	FT	68%	59%	82%	71%	81%	63%	72%	77%	67%
	Keys4BR	69%	55%	93%	74%	81%	67%	**74%**	**82%**	**67%**

the same results as the CNN, both of which have F1 scores 5% higher than that of SVM.

Furthermore, by comparing the results of Keys4BR with three semantic representation baseline techniques, namely VSM, PT, and FT, we find that the classifiers that perform best with Keys4BR in different tasks often also perform best with other semantic representation models (except on the bug reopen prediction task that for Keys4BR, SVM is optimal, while for others, CNN is best). The consistent performance of different classifiers across various semantic representation methods, to some extent, provides guidance for developers in selecting the most suitable classifiers for their specific types of bug report management tasks. For example, SVM would be their optimal classifier for bug priority prediction, and CNN is more preferred for bug severity prediction and bug reassignment prediction tasks.

In all four downstream bug report management tasks, Keys4BR demonstrates superior classification performance compared to the baseline methods of VSM, PT and FT, with average F1 score improvements of 8%, 13%, and 9%, respectively. Notably, under the optimal classifiers corresponding to each task, Keys4BR achieved higher F1 scores than the baselines of VSM, PT, and FT, further substantiating its superiority. Specifically, in the bug severity prediction task, although the PT and FT baselines achieved a high F1 score of 90% using the optimal CNN classifier, Keys4BR still managed to increase it by 1%. In the bug priority prediction task, with SVM as the classifier, the Keys4BR F1 score was increased by 6%, 19%, and 13% compared to the baselines VSM, PT and FT, respectively. In the bug reopen prediction task, using SVM as the classifier, the Keys4BR F1 score was elevated by 9%, 14%, and 12% compared to the VSM, PT, and FT baselines, respectively. In the bug reassignment prediction task, based on the better-performing classification layer and CNN classifiers, Keys4BR's F1 score was improved by 2% to 4% compared to the PT and FT baselines. Compared to other tasks, the performance improvement brought by Keys4BR in the bug reassignment prediction task is relatively limited. Such a difference may lie in that other three tasks largely rely on the textual content of bug report itself (hence better textual semantic capture leads to better performance improvements); while for the bug reassignment prediction task which aims to predict whether the values of certain meta items would get reassignment, the role of textual content is relatively weaker, other factors like reporters' misbehavior in filling these values may contributed larger to the reassignment results.

We further perform a statistical analysis using the Wilcoxon Rank Sum test [51] along with Cliff's Delta effect size [52] to assess whether the performance (F1 scores) differences observed between Keys4BR and the three semantic representation baselines are statistically significant, and to quantify the magnitude of these differences. Table 5 presents the statistical results across all downstream tasks, obtained by running the statistical tests on the pooled F1 scores of the four tasks. This aggregated analysis provides an overall view of the advantages of Keys4BR over the baselines in general bug report management scenarios. Table 6 complements this by examining the effect size for each individual task, offering a more fine-grained perspective on where Keys4BR achieves the most substantial improvements and where its benefits are less pronounced. This dual-level analysis – combining global and task-specific perspectives – ensures a more comprehensive and reliable interpretation of the performance differences. Note that for each task, since each technique has only three F1 score values, which makes the Wilcoxon test results unreliable [53], only Cliff's delta is reported in Table 6.

Table 5
Overall Wilcoxon rank-sum test results and Cliff's delta effect sizes in terms of F1 score differences between Keys4BR and three baselines.

Comparison pair	wilcoxon_pvalue	cliffs_delta	Difference magnitude
Keys4BR vs VSM	0.25	0.625	large
Keys4BR vs PT	0.00049	0.368	medium
Keys4BR vs FT	0.00049	0.299	small

The overall comparison results in Table 5 indicate that Keys4BR significantly outperforms VSM, PT, and FT in most cases, with p-values less than 0.001 and effect sizes ranging from small to large. Specifically, Keys4BR has a larger effect size compared to VSM (Cliff's $\delta = 0.625$), while the effect sizes are moderate when compared to PT ($\delta = 0.368$) and FT ($\delta = 0.299$).

From Table 6, we can find that Keys4BR achieved better performance than the baseline methods across all tasks, particularly in the priority and reopen tasks, where Cliff's δ values all reached 1 against VSM, PT, and FT, except for the 0.667 against FT in the reopen task, indicating a large effect size. In the reassignment task, Keys4BR also reached a large performance difference compared with VSM (Cliff's $\delta = 1.000$), PT, and FT (Cliff's $\delta = 0.556$ for PT and FT), as measured by Cliff's delta effect size. In the severity task, according to Cliff's δ, the difference magnitudes between Keys4BR and VSM, PT, and FT are large, medium, and medium, respectively. This suggests that, despite variations in the degree of performance difference across different tasks, Keys4BR consistently outperformed three baselines.

5.2. RQ2. What impact do different key sentences selection strategies have on the performance of Keys4BR?

To better understand which strategy can extract key information that is more helpful for the model to learn the representation of bug

Table 6
Cliff's delta effect sizes in terms of F1 score differences between Keys4BR and three baselines within each BRM task.

Task	Comparison Pair	cliffs_delta	Difference Magnitude
Bug Severity Prediction	Keys4BR vs VSM	1.000	large
	Keys4BR vs PT	0.444	medium
	Keys4BR vs FT	0.444	medium
Bug Priority Prediction	Keys4BR vs VSM	1.000	large
	Keys4BR vs PT	1.000	large
	Keys4BR vs FT	0.667	large
Bug Reopen Prediction	Keys4BR vs VSM	1.000	large
	Keys4BR vs PT	1.000	large
	Keys4BR vs FT	1.000	large
Bug Reassignment Prediction	Keys4BR vs VSM	1.000	large
	Keys4BR vs PT	0.556	large
	Keys4BR vs FT	0.556	large

report text, we also compare the strategy of selecting sentences that contain at least n localization keywords (i.e., At Least Strategy), the strategy of selecting the top k sentences with the highest number of keywords (i.e., Top Strategy), and the stack trace processing strategy. The details of separate selection strategies are below. Note that during the analysis, we used the SVM as the default classifier for Keys4BR fine-tuned using AdaLoRA, given its overall superior performance in RQ1. For consistency, we adopted a linear kernel with the regularization parameter C set to its default value, avoiding extensive hyperparameter tuning to ensure fair and efficient comparison between different key sentences selection strategies. The details are as follows.

(1) Strategy of selecting sentences containing at least n localization keywords (At Least Strategy): This strategy sets a threshold for the number of keywords contained in the sentence, and the sentence is considered as a key sentence only when the number of keywords in it reaches this threshold n. In this research question, the performance variation of Keys4BR in classification tasks will be investigated when the threshold n takes different values. Table 7 shows the performance changes of Keys4BR when different n values (i.e., 1, 2, 3) are used in the At Least Strategy. As can be seen from Table 7, as the n value gradually changes from 1 to 3, the performance of Keys4BR on the four downstream tasks shows a trend of first increasing and then decreasing. Among them, when the n value is 2, Keys4BR performs the best. When the n value is 3, Keys4BR performs better on priority prediction, reopen prediction, and reassignment prediction tasks than when the n value is 1. This indicates that when the n value is too small, the selected key sentences may still contain noise, which can affect the model during the learning process, leading to a biased representation of the bug report text.

(2) Strategy of selecting the top k sentences with the highest number of keywords (Top Strategy): Unlike the At Least Strategy which sets a minimum threshold for the number of keywords within a sentence, the Top Strategy limits the final number of key sentences selected. This strategy ranks sentences based on the number of keywords they contain and selects the top k sentences with the highest keyword counts as key information. In this study, we also investigated the performance of Keys4BR on four downstream tasks when the number of key sentences, k, takes on different values (i.e., 1, 2, 3). Table 8 shows the performance changes of Keys4BR under different k values for the Top Strategy. From the table, it can be observed that in the severity prediction and reassignment prediction tasks, Keys4BR performs best when k is 1. Meanwhile, in the priority prediction and reopen prediction tasks, Keys4BR performs better when k is 2. In addition, by comparing Tables 7 and 8, it can be found that under the At Least Strategy, the performance of Keys4BR is superior to that under the Top Strategy. This suggests that when selecting key information, setting a limit on the number of keywords can help select information that better promotes the model's ability to represent the text of bug reports. Therefore, compared to the Top Strategy, using the At Least Strategy can lead to better performance of the model.

Table 7
The performance variations of Keys4BR with different values of n in the At Least Strategy.

Task	n Value	F1 Score	Precision	Recall
Bug Severity Prediction	1	78%	69%	89%
	2	79%	69%	92%
	3	75%	74%	77%
Bug Priority Prediction	1	57%	57%	57%
	2	61%	61%	61%
	3	58%	58%	58%
Bug Reopen Prediction	1	74%	63%	91%
	2	82%	79%	85%
	3	80%	86%	75%
Bug Reassignment Prediction	1	61%	83%	49%
	2	69%	55%	93%
	3	63%	81%	51%

(3) Strategy of Processing Stack Trace: To ensure that model learning is not adversely affected by stack trace characteristics, we propose three stack trace processing strategies: directly removing stack trace (abbreviated as Removal), replacing the stack trace with the corresponding exception/error name (abbreviated as Replacement), and rewriting the stack trace using the predefined template (abbreviated as Rewriting) (detailed in Section 3.1.2). Table 9 shows the performance of Keys4BR in the four downstream tasks when applying each of these three stack trace processing strategies. For comparison purposes, the performance of the model without any stack trace processing (abbreviated as Origin) is also included. The results in the table indicate that Keys4BR performs best when using the name replacement strategy, followed by the strategy of directly deleting stack trace from the bug report text. This suggests that the special format of stack trace is likely to have a negative impact on the model's learning, leading to poor performance in the final downstream tasks. Therefore, it is necessary to appropriately process stack trace in the bug report text.

The comparative experimental results clearly demonstrate that the key sentences selection strategies and stack trace processing strategies proposed in this study have had a positive impact on how the model learns to represent bug report text. These strategies not only enhance the model's ability to capture, understand, and represent key information in complex text but also effectively reduce the interference caused by noise in bug reports, thereby improving the model's focus on core semantic information.

Table 8
The performance variations of Keys4BR with different values of k in the Top Strategy.

Task	k Value	F1 Score	Precision	Recall
Bug Severity Prediction	1	78%	75%	82%
	2	74%	59%	99%
	3	76%	85%	69%
Bug Priority Prediction	1	48%	48%	48%
	2	59%	59%	59%
	3	55%	55%	55%
Bug Reopen Prediction	1	79%	84%	75%
	2	82%	95%	72%
	3	80%	84%	77%
Bug Reassignment Prediction	1	67%	50%	100%
	2	64%	79%	53%
	3	62%	83%	51%

Table 9
The performance of Keys4BR on different stack trace processing strategies.

Task	Strategy	F1 Score	Precision	Recall
Bug Severity Prediction	Origin	77%	74%	80%
	Removal	77%	75%	79%
	Replacement	79%	69%	92%
	Rewriting	75%	73%	78%
Bug Priority Prediction	Origin	52%	52%	52%
	Removal	59%	59%	59%
	Replacement	61%	61%	61%
	Rewriting	58%	58%	58%
Bug Reopen Prediction	Origin	78%	82%	75%
	Removal	80%	80%	81%
	Replacement	82%	79%	54%
	Rewriting	80%	78%	82%
Bug Reassignment Prediction	Origin	61%	58%	65%
	Removal	66%	63%	70%
	Replacement	69%	55%	93%
	Rewriting	62%	60%	64%

5.3. RQ3. What impact do different lightweight fine-tuning strategies have on the performance of Keys4BR?

The fine-tuning strategy used by the model is also one of the key factors affecting the performance of Keys4BR. Therefore, it is necessary to explore and compare different fine-tuning strategies. In this study, three different lightweight fine-tuning strategies are compared: the frozen bottom-layer fine-tuning strategy (abbreviated as Frozen), the LoRA strategy, and the AdaLoRA strategy (with AdaLoRA being the strategy chosen for Keys4BR). This research question also uses a controlled variable approach to compare the performance of Keys4BR when using different fine-tuning strategies. That is, similar to RQ2, we used the SVM (with a linear kernel and default value setting for its regularization parameter C) as the classifier for Keys4BR; the key sentences strategy used is the At Least Strategy with n=2, and name replacement strategy for processing Stack Trace (as they have been found to perform best in RQ2).

As shown in Table 10, among the three different fine-tuning strategies, the AdaLoRA strategy performs the best, while the other two strategies result in slightly worse performance. Specifically, in the bug severity prediction task, when using the AdaLoRA strategy, the model's F1 score is the same as when using the LoRA strategy, and is improved by 3% compared to using the Frozen strategy. In the bug priority prediction task, the model's F1 score under the AdaLoRA strategy is 2% higher than when using the Frozen fine-tuning strategy, and 3% higher than when using the LoRA strategy. In the bug reopen prediction task, the model's F1 score under the AdaLoRA strategy is 7% higher than when using the Frozen strategy, and 6% higher than when using the LoRA strategy. In the bug reassignment prediction task, when using the AdaLoRA strategy, the model's F1 score is improved by 4% compared to when using the Frozen strategy, and by 1% compared to when using the LoRA fine-tuning strategy.

Table 10
The performance of Keys4BR after applying different fine-tuning strategies.

Task	Strategy	F1 Score	Precision	Recall
Bug Severity Prediction	Frozen	76%	76%	77%
	LoRA	79%	77%	81%
	AdaLoRA	79%	69%	92%
Bug Priority Prediction	Frozen	59%	59%	59%
	LoRA	58%	58%	58%
	AdaLoRA	61%	61%	61%
Bug Reopen Prediction	Frozen	75%	79%	71%
	LoRA	76%	74%	79%
	AdaLoRA	82%	79%	85%
Bug Reassignment Prediction	Frozen	65%	60%	71%
	LoRA	68%	58%	82%
	AdaLoRA	69%	55%	93%

The comparative results mentioned above indicate that for the area of bug report management, the AdaLoRA integrated into our Keys4BR is indeed more suitable for fine-tuning large language models, compared to another two commonly-used strategies. With AdaLoRA, our Keys4BR could perform best in understanding and representing the semantics of bug reports across four downstream bug report management tasks. The relatively better improvements of AdaLoRA over LoRA and Frozen strategies may lie in that AdaLoRA can adaptively adjust the updates of model parameters and flexibly allocate resources according to the specific needs of each task. This flexibility enables the AdaLoRA strategy to effectively enhance the model's semantic representation capabilities when dealing with different types of text classification tasks, thereby improving the model's performance. Its ability to reduce computational cost while maintaining model performance during the fine-tuning process is of great significance for resource optimization in practical applications.

6. Discussion

In this section, we mainly discuss the potential and practical considerations of Keys4BR, as well as the threats to the validity of our study. Details are as follows.

6.1. Potential and practical considerations of Keys4BR

Lightweight, Interpretable Fine-Tuning under Bug Reporting Noise. Keys4BR achieves a lightweight design by updating only a small subset of parameters during fine-tuning, which lowers computational requirements. Its interpretability stems from using bug-localizing keywords to extract contextually complete, bug-relevant information. By leveraging denoised, high-quality bug information, Keys4BR consistently outperforms mainstream baselines such as VSM-based methods or direct fine-tuning on raw reports of varying quality, demonstrating its effectiveness, efficiency, and practical value in noisy environments—even when only small amounts of data are available.

Offline, One-Time Keyword Extraction Enabling Efficient Deployment. The genetic-algorithm-based keyword extraction can be performed offline and only needs to be done once. It can also be parallelized across bug reports, making it a practical and non-bottleneck step. Organizations or individuals with moderate computational resources can collect a batch of bug–code data and fine-tune the model for general application. Once fine-tuned, the model can be directly used on new projects without repeating large-scale keyword extraction. Additionally, optional offline extraction on a subset of historical reports allows further project-specific fine-tuning, enhancing adaptability while maintaining low resource requirements.

Influence of Project-Specific Historical Data on Keys4BR's Adaptation. Although Keys4BR is designed as a pre-trained model that can

be applied to any project – even in the absence of historical bug reports and fixes – its performance can benefit from further project-specific fine-tuning when historical data is available. Leveraging such data allows Keys4BR to better adapt to the unique characteristics, patterns, and distributions of bugs in a given project, potentially improving the quality of its semantic representations.

Keyword Extraction Sensitivity to Bug Report Quality. Keys4BR relies on the quality of the keywords extracted from bug reports as its initial input. If the bug reports contain substantial noise or lack sufficient detail, the extracted keywords may not accurately reflect the underlying software issue, and in some cases, the genetic algorithm may not be able to extract any qualified keywords. This, in turn, could limit the effectiveness of the fine-tuned model. Fortunately, prior work [29] that introduced the keyword extraction method adopted in our study has shown that over 90% of bug reports from popular open-source projects yield high-quality keywords. Nevertheless, we acknowledge that this may vary in other contexts.

6.2. Threats to validity

Internal Validity. Our study faces four primary threats to internal validity. The first stems from the genetic algorithm and heuristic rules used for keyword extraction. Although we followed established practices [29], the algorithm's performance is sensitive to hyperparameters such as population size and mutation rate, and any sub-optimal tuning could degrade the quality of the extracted keywords.

A second threat arises from the data-balancing procedure. Synonym replacement and under-sampling may inject synthetic noise or discard informative instances, thereby influencing classifier behavior. While we compared several balancing strategies and retained only the best-performing configuration, the intrinsic bias introduced by these techniques remains a potential limitation.

A third threat lies in the choice of AdaLoRA as the fine-tuning strategy. Our decision was guided by empirical comparisons against Frozen, LoRA, and full fine-tuning, yet we cannot guarantee that AdaLoRA is universally superior to other PEFT variants or to full fine-tuning on unseen tasks or models. We acknowledge this limitation and encourage future studies to re-evaluate our conclusions under alternative fine-tuning regimes.

A fourth threat concerns the quantification of Keys4BR's computational cost. In this study, we mainly provided a qualitative clarification of the efficiency of keyword extraction and fine-tuning based on their underlying principles and some supportive evidence from prior studies. Due to insufficient prior planning, some quantitative measurements, such as per-report keyword extraction time, fine-tuning time for AdaLoRA versus LoRA versus full fine-tuning, or peak memory usage, were not collected during the experiments. We recognize this as a limitation and suggest that future work systematically assess these computational costs.

External Validity. The generalizability of Keys4BR is threatened by several factors. First, our study was conducted on five Java projects that are frequently used in the research community for bug report handling tasks [46–49], facilitating alignment and comparability with prior work. Although these projects span different domains (e.g., aspect-oriented programming, business intelligence and reporting, and user interface frameworks) and vary in size and functionality, providing a certain degree of diversity, the findings may still not fully generalize to projects developed in other languages, operating in different domains, or adhering to distinct development cultures.

Second, our evaluation was restricted to four BRM tasks: severity, priority, reopen, and reassignment prediction. Whether Keys4BR remains effective in tasks such as duplicate detection or assignee recommendation remains to be verified. We therefore plan to broaden the task spectrum in subsequent studies.

Finally, our findings are tied to the specific combination of RoBERTa as the base LLM and SVM/CNN/Classification Layer as downstream classifiers. Alternative models (e.g., CodeBERT, DeBERTa) or classifiers (e.g., Random Forest, Transformer-based architectures) might produce divergent outcomes. While we observed consistent trends across multiple classifiers, we cannot preclude the possibility that other modeling choices could challenge our conclusions.

7. Conclusion

This paper introduces Keys4BR, a key sentences-based model fine-tuning for better semantic representation of bug reports. Initially, Keys4BR constructs a set of bug localization keywords using the genetic algorithm and heuristic rules for keyword optimization, which is adept at capturing the essential semantic information within bug reports. Subsequently, various key sentences selection strategies and lightweight fine-tuning techniques are employed to extract key information from bug reports and fine-tune the large language model, enabling them to better comprehend and represent the deep semantic information of bug reports. By comparing the classification performance of Keys4BR with several baseline methods across four bug report management tasks, the effectiveness of the proposed Keys4BR technique and the significance of its main technical components are validated. To further enhance the generalizability of the proposed technology, future work will involve validating the effectiveness of Keys4BR on additional bug report management tasks, such as bug report assignment and duplicate bug report detection, and refining Keys4BR based on task feedback.

CRediT authorship contribution statement

Mengjiao Wang: Writing – review & editing, Writing – original draft, Software, Methodology, Investigation. **Biyu Cai:** Software, Methodology, Investigation, Formal analysis, Data curation, Conceptualization. **Weiqin Zou:** Supervision, Project administration, Data curation, Conceptualization. **Jingxuan Zhang:** Supervision, Investigation, Conceptualization.

Code and data availability.

The code and datasets used in this study are available in the Keys4BR repository (https://github.com/wmmjj/Keys4BR.git).

Declaration of competing interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

Appendix. Performance comparison between Keys4BR and RepresentThemAll

In this part, we further conduct a comparison between our Keys4BR and the state-of-the-art semantic representation model RepresentThemAll [21] pretrained on a large dataset of over 200,000 raw bug reports. We include this comparison in the appendix to provide additional insight into how Keys4BR performs relative to a large-scale pretrained model, even though the two are trained on datasets of vastly different sizes (18,000- vs. 200,000+ bug reports) and cannot be aligned under identical fine-tuning conditions (due to the unavailability of the corresponding bug–code data for the 200,000 bug reports of RepresentThemAll).

Specifically, we first use Keys4BR and RepresentThemAll to separately generate semantic representations for the test bug reports of our projects, and then build and evaluate the classification models under different classifier choices for the four downstream bug report management tasks. Table A.11 presents the detailed performance of Keys4BR and RepresentThemAll (RTA).

Table A.11
The comparison results of Keys4BR and RepresentThemAll (RTA).

Task	Method	SVM			Classification layer			CNN		
		F1 score	Precision	Recall	F1 score	Precision	Recall	F1 score	Precision	Recall
Bug Severity Prediction	RTA	81%	71%	94%	92%	90%	93%	92%	90%	92%
	Keys4BR	79%	69%	92%	90%	89%	90%	91%	90%	91%
Bug Priority Prediction	RTA	62%	62%	62%	55%	55%	55%	50%	50%	50%
	Keys4BR	61%	61%	61%	52%	52%	52%	47%	47%	47%
Bug Reopen Prediction	RTA	83%	75%	92%	80%	78%	82%	81%	83%	79%
	Keys4BR	82%	79%	85%	79%	76%	82%	79%	81%	78%
Bug Reassignment Prediction	RTA	69%	67%	72%	76%	86%	68%	75%	84%	67%
	Keys4BR	69%	55%	93%	74%	81%	67%	74%	82%	67%

From the table, we can find that RTA slightly outperformed our Keys4BR in the four downstream tasks by 0–3 percent in terms of F1 scores. Our further Wilcoxon test results indicated that these observed performances have statistical significance at the p-value <0.001 (p-value = 0.00098). However, by further checking Cliff's delta effect size over the F1 scores of Keys4BR and RTA, we find that the observed performance differences are negligible, with a Cliff's delta effect size = −0.146.

RTA is a universal bug report framework pre-trained on over 200,000 raw, undenoised reports using two carefully designed objectives: a dynamic masked language model and a contrastive learning objective. In contrast, Keys4BR focuses on fine-tuning large models with denoised, high-quality bug information and designed a concrete and practical way of extracting key bug-related information to denoise raw bug reports. By using a much smaller dataset for fine-tuning, Keys4BR achieved performance comparable to RTA, demonstrating its unique advantages of being lightweight, efficient, and interpretable in the "small data + noise" scenario. More importantly, the paradigm of using denoised, high-quality bug information to enhance LLM fine-tuning is generalizable. We believe that applying denoised key information to RTA could further enhance its performance, while scaling up the fine-tuning dataset for Keys4BR would also improve its results. Given that our primary contribution lies in proposing the denoising-based fine-tuning paradigm, we leave the use of larger-scale denoised datasets to fine-tune Keys4BR to future organizations or researchers interested in deploying and extending Keys4BR.

Data availability

Data will be made available on request.

References

[1] Z. Luo, W. Wang, C. Cen, Improving bug localization with effective contrastive learning representation, IEEE Access 11 (2023) 32523–32533.
[2] S. Kim, E.J. Whitehead, How long did it take to fix bugs? in: Proceedings of the 2006 International Workshop on Mining Software Repositories, 2006, pp. 173–174.
[3] B. Chen, W. Zou, B. Cai, et al., An empirical study on the potential of word embedding techniques in bug report management tasks, Empir. Softw. Eng. 29 (2024) 122.
[4] Y. Tan, S. Xu, Z. Wang, T. Zhang, Z. Xu, X. Luo, Bug severity prediction using question-and-answer pairs from stack overflow, J. Syst. Softw. 165 (2020) 110567.
[5] Y. Tian, D. Lo, X. Xia, C. Sun, Automated prediction of bug report priority using multi-factor analysis, Empir. Softw. Eng. 20 (2015) 1354–1383.
[6] S. Mani, A. Sankaran, R. Aralikatte, Deeptriage: Exploring the effectiveness of deep learning for bug triaging, in: Proceedings of the ACM India Joint International Conference on Data Science and Management of Data, 2019, pp. 171–179.
[7] B. Xiang, Y. Shao, SumLLaMA: Efficient contrastive representations and fine-tuned adapters for bug report summarization, IEEE Access 12 (2024) 78562–78571.
[8] J. Devlin, M.-W. Chang, K. Lee, K. Toutanova, BERT: Pre-training of deep bidirectional transformers for language understanding, in: Proceedings of the 2019 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, 2019, pp. 4171–4186.
[9] W.-Y. Wang, C.-H. Wu, J. He, Clebpi: Contrastive learning for bug priority inference, Inf. Softw. Technol. 164 (2023) 107302.
[10] A. Ali, Y. Xia, Q. Umer, M. Osman, Bert based severity prediction of bug reports for the maintenance of mobile applications, J. Syst. Softw. 208 (2024) 111898.
[11] C. Zhou, P. Liu, P. Xu, S. Iyer, J. Sun, Y. Mao, X. Ma, A. Efrat, P. Yu, L. Yu, et al., Lima: Less is more for alignment, in: Advances in Neural Information Processing Systems, vol. 36, 2024.
[12] A.K. Dipongkor, K. Moran, A comparative study of transformer-based neural text representation techniques on bug triaging, in: 2023 38th IEEE/ACM International Conference on Automated Software Engineering, 2023, pp. 1012–1023.
[13] W. Zou, D. Lo, Z. Chen, X. Xia, Y. Feng, B. Xu, How practitioners perceive automated bug report management techniques, IEEE Trans. Softw. Eng. 46 (2020) 836–862.
[14] D. Huo, T. Ding, C. McMillan, M. Gethers, An empirical study of the effects of expert knowledge on bug reports, in: 2014 IEEE International Conference on Software Maintenance and Evolution, 2014, pp. 1–10.
[15] Z. Y., L. J.K., X. X., W. M.H., Y. H., Research progress on software bug localization technology based on information retrieval, J. Softw. 31 (2020) 2432–2452.
[16] J. Anvik, G.C. Murphy, Reducing the effort of bug report triage: Recommenders for development-oriented decisions, ACM Trans. Softw. Eng. Methodol. 20 (2011) 1–35.
[17] W.-Q. Zou, J.-X. Zhang, X.-W. Zhang, L. Chen, J.-F. Xuan, Survey of research on bug report quality, Ruan Jian Xue Bao/Journal Softw. (2023).
[18] K. Somasundaram, G.C. Murphy, Automatic categorization of bug reports using latent Dirichlet allocation, in: Proc. of the 5th India Software Engineering Conf., 2012, pp. 125–130.
[19] D.M. Blei, A.Y. Ng, M.I. Jordan, Latent dirichlet allocation, J. Mach. Learn. Res. 3 (2003) 993–1022.
[20] T. Mikolov, I. Sutskever, K. Chen, G.S. Corrado, J. Dean, Distributed representations of words and phrases and their compositionality, in: Advances in Neural Information Processing Systems, vol. 26, 2013.
[21] S. Fang, T. Zhang, Y. Tan, H. Jiang, X. Xia, X. Sun, RepresentThemAll: A universal learning representation of bug reports, in: 2023 IEEE/ACM 45th International Conference on Software Engineering, 2023, pp. 602–614.
[22] S. Fang, Y.-s. Tan, T. Zhang, Z. Xu, H. Liu, Effective prediction of bug-fixing priority via weighted graph convolutional networks, IEEE Trans. Reliab. 70 (2021) 563–574.
[23] H.M. Tran, S.T. Le, S. van Nguyen, P.T. Ho, An analysis of software bug reports using machine learning techniques, SN Comput. Sci. 1 (2020) 4.
[24] W.Y. Ramay, Q. Umer, X.C. Yin, C. Zhu, I. Illahi, Deep neural network-based severity prediction of bug reports, IEEE Access 7 (2019) 46846–46857.
[25] J. Kim, G. Yang, Bug severity prediction algorithm using topic-based feature selection and CNN-LSTM algorithm, IEEE Access 10 (2022) 94643–94651.
[26] E. Shihab, A. Ihara, Y. Kamei, et al., Studying re-opened bugs in open source software, Empir. Softw. Eng. 18 (2013) 1005–1042.
[27] X. Xia, D. Lo, E. Shihab, X. Wang, B. Zhou, Automatic, high accuracy prediction of reopened bugs, Autom. Softw. Eng. 22 (2015) 75–109.
[28] X. Xia, D. Lo, E. Shihab, X. Wang, Automated bug report field reassignment and refinement prediction, IEEE Trans. Reliab. 65 (2016) 1094–1113.
[29] B. Cai, W. Zou, Q. Meng, et al., KBL: a golden keywords-based query reformulation approach for bug localization, Empir. Softw. Eng. 30 (2025) 135.
[30] N. Bettenburg, S. Just, A. Schröter, C. Weiss, R. Premraj, T. Zimmermann, What makes a good bug report? in: Proceedings of the 16th ACM SIGSOFT International Symposium on Foundations of Software Engineering, 2008, pp. 308–318.
[31] E.J. Hu, Y. Shen, P. Wallis, Z. Allen-Zhu, Y. Li, S. Wang, L. Wang, W. Chen, Lora: Low-rank adaptation of large language models, 2021, arXiv preprint arXiv:2106.09685.
[32] Q. Zhang, M. Chen, A. Bukharin, N. Karampatziakis, P. He, Y. Cheng, W. Chen, T. Zhao, AdaLoRA: Adaptive budget allocation for parameter-efficient fine-tuning, 2023, arXiv preprint arXiv:2303.10512.
[33] Z. Han, C. Gao, J. Liu, J. Zhang, S.Q. Zhang, Parameter-efficient fine-tuning for large models: A comprehensive survey, Trans. Mach. Learn. Res. 2024 (2024).

[34] Y. Liu, M. Ott, N. Goyal, J. Du, M. Joshi, D. Chen, O. Levy, M. Lewis, L. Zettlemoyer, V. Stoyanov, RoBERTa: A robustly optimized BERT pretraining approach, 2019, arXiv preprint arXiv:1907.11692.

[35] M.A. Hadi, F.H. Fard, Evaluating pre-trained models for user feedback analysis in software engineering: a study on classification of app-reviews, Empir. Softw. Engg. 28 (2023).

[36] C. Raffel, N.M. Shazeer, A. Roberts, K. Lee, S. Narang, M. Matena, Y. Zhou, W. Li, P.J. Liu, Exploring the limits of transfer learning with a unified text-to-text transformer, J. Mach. Learn. Res. 21 (2019) 140:1–140:67.

[37] J. Ni, G.H. Abrego, N. Constant, J. Ma, K. Hall, D. Cer, Y. Yang, Sentence-T5: Scalable sentence encoders from pre-trained text-to-text models, in: Findings of the Association for Computational Linguistics, 2022, pp. 1864–1874.

[38] P. He, X. Liu, J. Gao, W. Chen, DeBERTa: Decoding-enhanced BERT with disentangled attention, 2020, arXiv preprint arXiv:2006.03654.

[39] P. He, J. Gao, W. Chen, DebertaV3: Improving deberta using electra-style pre-training with gradient-disentangled embedding sharing, 2021, arXiv preprint arXiv:2111.09543.

[40] J.C. Timoneda, S.V. Vera, BERT, roberta, or deberta? Comparing performance across transformers models in political science text, J. Politics 87 (2025) 347–364.

[41] A. Lamkanfi, S. Demeyer, E. Giger, B. Goethals, Predicting the severity of a reported bug, in: Proceedings of the 2010 7th IEEE Working Conference on Mining Software Repositories, 2010, pp. 1–10.

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

[43] M. Alenezi, S. Banitaan, Bug reports prioritization: which features and classifier to use? in: Proceedings of the 12th International Conference on Machine Learning and Applications, 2013, pp. 112–116.

[44] M. Izadi, K. Akbari, A. Heydarnoori, Predicting the objective and priority of issue reports in software repositories, Empir. Softw. Eng. 27 (2022) 50.

[45] X. Ye, R.C. Bunescu, C. Liu, 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, 2014, pp. 689–699.

[46] A.N. Lam, A.T. Nguyen, H.A. Nguyen, T.N. Nguyen, Bug localization with combination of deep learning and information retrieval, in: 2017 IEEE/ACM 25th International Conference on Program Comprehension, 2017, pp. 218–229.

[47] S. Shao, T. Yu, Information retrieval-based fault localization for concurrent programs, in: 2023 38th IEEE/ACM International Conference on Automated Software Engineering, 2023, pp. 1467–1479.

[48] Y. Xiao, J. Keung, Q. Mi, K.E. Bennin, Improving bug localization with an enhanced convolutional neural network, in: 2017 24th Asia-Pacific Software Engineering Conference, 2017, pp. 338–347.

[49] X. Xiao, R. Xiao, Q. Li, J. Lv, S. Cui, Q. Liu, BugRadar: Bug localization by knowledge graph link prediction, Inf. Softw. Technol. 162 (2023) 107274.

[50] C. Mills, E. Parra, J. Pantiuchina, et al., On the relationship between bug reports and queries for text retrieval-based bug localization, Empir. Softw. Eng. 25 (2020) 3086–3127.

[51] H.B. Mann, D.R. Whitney, On a test of whether one of two random variables is stochastically larger than the other, Ann. Math. Stat. 18 (1947) 50–60.

[52] G. Macbeth, E. Razumiejczyk, R. Ledesma, Cliff's delta calculator: A non-parametric effect size program for two groups of observations, Univ. Psychol. 10 (2011) 545–555.

[53] Y. Cao, et al., Why is a small sample size not enough? Oncol. 29 (2024) 761–763.