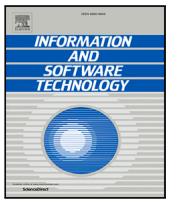
Information and Software Technology 181 (2025) 107675

Contents lists available at ScienceDirect

Information and Software Technology

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



A more accurate bug localization technique for bugs with multiple buggy code files

Hui Xu [a,1], Zhaodan Wang [a,1], Weiqin Zou [a,b,*]

[a] College of Computer Science and Technology, Nanjing University of Aeronautics and Astronautics, Nanjing, China
[b] State Key Laboratory for Novel Software Technology, Nanjing University, Nanjing, China

ARTICLE INFO

Dataset link: https://github.com/LyraXv/HitMore

Keywords:
Bug localization
Multiple buggy files
Subset retrieval
Code relations

ABSTRACT

Context: Bug localization is a key step in bug fixing. Despite considerable progress, existing bug localization techniques still perform unsatisfactorily in situations where the complete fix to a bug involves touching multiple buggy code files. That is, for such bugs, those techniques tend to locate correctly only one or at least not all buggy code files, leaving other buggy code files undetected.
Objective: This study aims to improve bug localization in cases where resolving a bug requires modifications to multiple buggy code files by proposing HitMore to rank more truly buggy files higher in the recommendation list.
Method: The basic idea of HitMore is to attempt to retrieve a subset of truly buggy code files first, then use these files to retrieve other buggy code files based on code relation analysis. For the first part, we designed three kinds of domain-specific features to build a machine-learning model to identify the truly buggy code file subset. For the second part, we make use of three types of code relations between the code base and the buggy file subset to better retrieve the remaining truly buggy code files.
Results: The experiments on six widely open-source projects show that: Our technique is effective in identifying the subset of truly buggy code files, with a weighted prediction F1-Score of 86.1%–92.1%. By leveraging the code relations to the retrieved subset and the code base, our HitMore could retrieve all truly buggy code files for 29.31%–69.56% of bugs across six projects. For multiple-buggy-code-file bugs, HitMore could completely localize such bugs by up to 15.38%, 19.36%, and 11.86% more than three representative IRBL baselines across six projects.
Conclusion: The experimental results demonstrate the potential of HitMore in reducing developers' burden of locating and further fixing relatively complex bugs such as those with multiple buggy code files in practice.

1. Introduction

Bug localization is an important part of bug repair [1,2]. In order to facilitate bug repair and ensure the quality of products, researchers have proposed various bug localization techniques which can be classified into two categories: dynamic bug localization and static bug localization [3–9]. Dynamic bug localization techniques aim to locate bugs by analyzing execution profiles collected while running the to-be-debugged software products [3] . Static bug localization techniques do not need to run the buggy program; instead, they mainly extract some static semantic features from the code and bug reports, do some semantic matching (e.g., based on cosine similarity), and then output a recommendation list of buggy code elements [10]. With comparative performance and the cost reduction of no program running over dynamic bug localization, static bug localization receives much attention from both the academic and industrial communities. The main-stream and also the most representative of static bug localization is the information retrieval-based (IR-based) bug localization [11–16]. IR-based bug localization usually considers bug localization as an information retrieval task, which generates a prioritized list of suspicious code files for a bug report based on their static semantic relevance scores [8] (kinda like the scenario of searching Google for relevant documents). In this paper, we also focus on developing advanced static bug localization techniques, specifically, the more advanced IR-based bug localization techniques.

* Corresponding author.
E-mail addresses: lyraxv@nuaa.edu.cn (H. Xu), wangzhaodan@nuaa.edu.cn (Z. Wang), weiqin@nuaa.edu.cn (W. Zou).
[1] Hui Xu and Zhaodan Wang are both first authors, and contributed equally to the work.

https://doi.org/10.1016/j.infsof.2025.107675
Received 4 March 2024; Received in revised form 30 November 2024; Accepted 21 January 2025
Available online 31 January 2025

Despite considerable advancements, existing IR-based bug localization (IRBL) techniques fail to handle multiple-buggy-code-file bugs effectively. Here, a multiple-buggy-code-file bug refers to a bug for which the complete fix requires modifying multiple code files. In practice, it is not uncommon for developers to handle the multiple-buggy-code-file bugs. For instance, based on our preliminary analysis of 5297 bugs in six open-source software projects (i.e., Tomcat, AspectJ, Lucene, ZooKeeper, OpenJPA, and Hibernate-ORM) in this paper, there are 3215 bugs each of which has at least two corresponding buggy code files. Nevertheless, by analyzing the localization results (with a recommendation list size of 5) of three mainstream IR-based bug localization techniques (i.e., Amalgam [14], Blizzard [17], and BugLocator [12]), it can be found that many buggy code files for these bugs could not be retrieved correctly (the three techniques missed buggy code files for 2812/2799/2759 bugs).

One potential cause of the problem may lie in how current IRBL techniques operate. Typically, the majority of IRBL techniques [8,15–17] tend to treat each code file as an isolated entity, calculating its similarity to a bug report independently and ranking files based on this similarity score in a unified output list. However, for multiple-buggy-code-file bugs, the involved buggy files often have interdependencies. While existing IRBL techniques can often ensure at least one buggy file appears in the recommendation list, treating each file independently and overlooking these relationships may make it difficult to identify all relevant buggy files within the same list accurately.

Towards this problem, we propose developing a new bug localization technique namely HitMore. The idea of HitMore is inspired by two considerations. One consideration is that mainstream static bug localization techniques have shown reasonably good performance in generating recommendation lists that, while not covering all buggy files, usually include at least one truly buggy file [12,15,17]; the problem is that the exact buggy file remains unknown. The other consideration is that the buggy code files of a bug are generally related by nature through, for example, data dependencies or other relations. If we can obtain a subset of truly buggy code files, then it is likely for us to retrieve other buggy code files scattered in the code base through relation analysis.

Hence, we propose to develop HitMore, which builds a classifier to identify a subset of truly buggy files from recommendation lists first (support by the first consideration), then use the subset to retrieve the remaining buggy files through leveraging three kinds of code relations (support by the second consideration). In the subset identification step, we construct three kinds of domain-specific features of bug reports and codes, and then build a classifier to predict which code file within the recommendation list is truly buggy. Then, we leverage three kinds of code relations including control flow dependency, data flow dependency, and co-occurrence dependency, to retrieve the remaining truly buggy code files.

A dataset of six widely used open-source Java projects is used to evaluate our HitMore. The experimental results indicate that HitMore could achieve a good performance in recognizing a subset of truly buggy files, with weighted prediction Accuracy of 85.5%–92.7%, Precision of 86.7%–91.7%, Recall of 85.5%–92.7%, and F1-Score of 86.1%–92.1%. The importance of three kinds of features for truly buggy code files identification is further analyzed to better understand our technique. Overall, HitMore could retrieve all truly buggy code files for 29.31%–69.56% of bugs across six projects. For multiple-buggy-code-file bugs, HitMore could completely localize such bugs by up to 15.38% (6.83% on average), 19.36% (6.18% on average), and 11.86% (7.42% on average) more than the three representative IRBL baselines (i.e., AmaLgam, BugLocator, and Blizzard) across six projects. The MAP and MRR for HitMore ranged from 0.35–0.58, and 0.43–0.72, which outperformed three baselines by 0.19/0.12/0.12 and 0.2/0.09/0.1 on average. The improvement in MAP indicates that our technique ranks more buggy files higher in the recommendation list, and the higher MRR reflects that it can position the truly buggy file closer to the top of the list. The results show that HitMore can more effectively help developers find multiple buggy code files corresponding to bugs at once.

The major contributions are as follows:

- We highlight the problem that a bug with multiple buggy files could not be well located by existing bug localization techniques and propose a more advanced solution.
- We develop a technique HitMore that aims to locate the multiple-buggy-code-files bugs by identifying a subset of truly buggy code files first and then use the subset to retrieve other remaining buggy code files through different code file relations.
- We construct a dataset of 5297 bugs to evaluate our HitMore and obtain promising results that verify the potential of our technique in handling bugs with multiple corresponding buggy code files.

The rest of this paper is organized as follows. Section 2 presents the background and related work of our study. Section 3 detailedly introduces our technique. Section 4 describes the experimental setup. Section 5 presents the experimental results. Section 6 discusses the threats to the validity of our study. Section 7 concludes our work.

2. Background and related work

Bug localization aims to support the debugging activities of developers by highlighting the code elements that are suspected to be responsible for the observed failure [4,18,19]. It allows developers to concentrate on vital files [20]. Existing bug localization techniques can be divided into two categories based on whether or not to run test cases, namely dynamic bug localization and static bug localization [3–10].

Dynamic bug localization tends to run tested programs under a set of test cases and leverage program execution traces to associate code elements with program failures [4]. Spectrum-based bug localization techniques are representatives of such techniques [3,21,22]. Static bug localization generally requires no 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 elements, where a program element with a higher semantic similarity with a bug report is considered more relevant to the bug [10]. Information retrieval-based bug localization (IRBL) techniques are representatives of such techniques [8,12,16,17,23]. As the focus of our study is also on IRBL techniques, we will mainly introduce the general idea of IRBL and related studies in this research area.

IRBL views bug localization as a document retrieval task: bug reports function as queries, code files act as the document database, and the goal is to retrieve the code files most relevant to a given bug report. Fig. 1 illustrates a real bug report from the Tomcat project. A typical bug report usually consists of textual information items like the bug title and detailed bug description, as well as non-textual meta-information items, including priority, severity, status, etc. In IRBL, bug reports and code files are typically treated as textual data, with their semantics extracted using traditional information retrieval techniques, such as the vector space model (VSM) [12,24], latent Dirichlet allocation (LDA) [25,26], and latent semantic indexing (LSI) [27,28]. Semantic vectors extracted from a bug report and a code element, for example using VSM, are then compared to calculate a similarity score, usually using cosine similarity. The code elements with the highest similarity scores are identified as potential buggy candidates for further inspection.

In the early stage, mainly the textual contents of bug reports and code files themselves are considered during semantic representation by IR models and subsequent similarity calculation [4,12,26,27] in bug localization. Considering that other information in the software may also help in bug localization, some researchers started to mine specific kinds of information related to bugs and code to improve bug-locating performance.

Bug 54379 - Implement support for post-construct and pre-destroy elements in web.xml

Status: RESOLVED FIXED
Alias: None
Product: Tomcat 7
Component: Catalina (show other bugs)
Version: 7.0.34
Hardware: PC Windows XP
Importance: P2 normal (vote)
Target Milestone: ---
Assignee: Tomcat Developers Mailing List
URL:
Keywords:
Depends on:
Blocks:

Reported: 2013-01-07 04:07 UTC by Konstantin Kolinko
Modified: 2013-01-10 11:48 UTC (History)
CC List: 0 users

Attachments

Patch proposal + tests (40.85 KB, patch) — Details | Diff
2013-01-10 07:32 UTC, Violeta Georgieva
Add an attachment (proposed patch, testcase, etc.) — View All

Note
You need to log in before you can comment on or make changes to this bug.

Konstantin Kolinko 2013-01-07 04:07:15 UTC — Description

There appears to be a feature that web[-fragment].xml file can contain such elements as <post-construct> and <pre-destroy>, and they are treated as equivalents to @PostConstruct and @PreDestroy annotations being present on the classes mentioned in them.

This feature is
1. mentioned in the Java EE 6 Platform Specification, chapter "EE.5.2.5 Annotations and Injection" [1]
2. mentioned in chapter "8.2.3 Assembling the descriptor from web.xml, webfragment.xml and annotations",
see points "k." and "l." on page 81 (103/230) of servlet-3_0-mrel-spec.pdf

[1] javaee_platform-6_0-fr-spec.pdf
http://jcp.org/about/java/communityprocess/final/jsr316/index.html

An example can be found in Jetty wiki:
http://wiki.eclipse.org/Jetty/Feature/Annotations#Lifecycle_callbacks:_PostConstruct_PreDestroy

Searching by the tag names, I do not see any code in the current trunk that processes those XML elements.

Violeta Georgieva 2013-01-07 20:27:41 UTC — Comment 1

Hi,

I'm going to work on this.

Regards
Violeta

Fig. 1. A bug report example with bugID = 54 379 in Tomcat Project.

Nguyen et al. [29] proposed BugScout, which considered the technical aspects in the textual contents of bug reports correlated with buggy files and customized LDA for bug localization. Sisman et al. integrate version history information into an IRBL framework [30]. Zhou et al. [12] proposed BugLocator leveraging similar bug reports for bug localization. Saha et al. [11] proposed BLUiR, which leverages structured information retrieval based on code constructs like class/method/variable names from bug reports and source files. Wang and Lo [14] integrated version history, bug reports and structured information to enhance the locating performance. To further improve localization results, they also incorporated stack traces and reporter information [10]. Considering the lexical gap between bug reports and code, Ye et al. [31] introduced word embedding techniques to project them as embedding vectors in the shared space. Xiao et al. [23] proposed BugRadar, which utilized text features and employed knowledge graphs to represent structure features from bug reports and source files.

Unlike the above studies, which focus on leveraging various potentially useful information for bug localization, some researchers propose to focus on the quality of bug reports themselves by improving their quality to facilitate bug localization. They developed several bug report reformulation strategies to reduce noise within bug reports or expand useful information to the original bug reports. Sisman et al. [32] performed an initial localization based on the original bug report to get a ranked list of relevant documents first, and then extracted additional words from the top-ranked documents to expand the original bug reports. Chaparro et al. [33] observed that low-quality bug reports contain irrelevant terms; they enhanced localization performance by refining them to include only observed behavior. Rahman et al. [17] proposed Blizaard, which categorized bug reports into three types and applied corresponding query reformulation to them to do bug localization. Kim and Lee [34] enhanced the quality of bug reports by utilizing attachments and reformulating bug reports by reducing noisy terms.

Some other researchers tried to apply deep/machine learning models to bug localization work. Lam et al. [35] first combined deep neural networks (DNNs) with rVSM for bug localization. Meng et al. [36] proposed TRANSFER, which utilized deep semantic features, as well as knowledge transferred from open-source data to improve bug localization and bug repair. Chakraborty et al. [9] proposed RLocator which adopted reinforcement learning to directly optimize evaluation measures. Yousofvand et al. [37] suggested that the bug localization problem could be treated as a node classification problem. They first used the Gumtree algorithm to label nodes by comparing the error graph with its corresponding repair graph, and then used a graph neural network for classification. Koyuncu et al. [38] proposed a multi-classifier approach to improve the performance of IRBL. Khatiwada et al. [39] proposed an optimized method that could systematically combine various IRBL techniques into hybrid pairs. Le et al. proposed APRILE [40], which introduced the ensemble method as a component for predicting whether the bug localization is effective.

Unlike previous studies, we pay special attention to the localization of multiple-buggy-code-file bugs, which existing IRBL studies to some extent overlooked and failed to locate well. Towards this kind of bugs, we proposed a new approach that identifies a subset of truly buggy code files first and then leverages the inherent dependency (three kinds of code relations) between truly buggy code files to retrieve the remaining truly buggy code files.

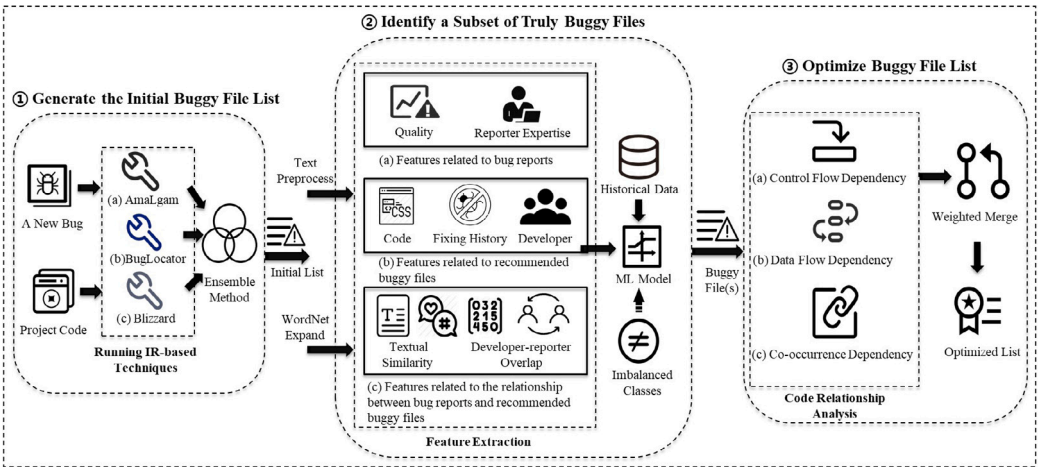


Fig. 2. Overview of HitMore.

3. Our HitMore approach

This section describes our technique in detail. We first describe the overall framework of our HitMore, and then introduce each component of HitMore.

3.1. Overview

As illustrated in Fig. 2, our technique contains three steps, including generating the initial buggy file list, identifying a subset of truly buggy files, and optimizing buggy file lists. The goal of the first step is to obtain an initial buggy file list that very likely contains at least one truly buggy file(s) so as to provide a good data basis for the second step to identify the exact subset of truly buggy code files. The retrieved subsets of truly buggy code files are then fed to the third step, where three code file lists having control flow dependency, data flow dependency, and co-occurrence dependency relations with the retrieved subsets of truly buggy code files, are obtained, merged, and output an optimized recommendation list that is expected to contain more truly buggy code files.

As mentioned in Section 1, the idea of HitMore is to precisely identify a subset of truly buggy code files first, then leverage different kinds of code relationships between truly buggy code files and remaining code files to retrieve other truly buggy files. The prerequisite for the success of subset retrieval of truly buggy code files is that the input list should, as much as possible, ensure the presence of truly buggy code files for retrieval, and this is the goal of the first step.

In the first step, three representative IRBL techniques are used to help obtain a good initial file list that largely contains truly buggy code files. Specifically, for a bug report, three IRBL techniques, including AmaLgam, Blizzard, and BugLocator, are applied first to retrieve a suspicious file list from the code base separately. Then, with the aim to enhance the likelihood of including truly buggy files in the final initial list, we leverage the complementary nature of (IR-based) bug localization techniques — by combining their individual suspicious file lists through an ensemble method [2,38]. This improves the overall accuracy by addressing the strengths and weaknesses of each IRBL baseline. Furthermore, as reported by existing studies, the three techniques generally achieved an Accuracy@10 (accuracy in hitting at least one buggy file when recommending 10 files) in the range of 60%+ to 80%+ across different datasets [17,38], which are reasonably good but may still not be good enough for our first step. To further increase the likelihood of a list containing truly buggy code file(s), we decide to use the recommendation lists of size 20 by three IRBL techniques in the first step.

In the second step, we take the subset retrieval of truly buggy code files from the initial list output by the first step as a machine learning task. More specifically, a classification model is first built on historical bug report-code file data, where for each pair of a bug report and a suspicious code file, three kinds of features related to the bug report, the suspicious code file, and their relationships are collected; and the class label is assigned 1 if the suspicious code file is truly buggy for the bug report, otherwise 0. For each new arrival bug report and its initial buggy file list, the subset retrieval of truly buggy code files is then transformed to use the classification model to predict which suspicious code files are truly buggy code files based on corresponding instance feature values.

In the third step, we use the obtained subset to retrieve the remaining buggy code files. This is accomplished by analyzing the subset and code base through control flow, data flow, and co-occurrence analysis. We then employ a weighted strategy to merge and reorder the code files, which helps to generate an optimized list of buggy code files. If the subset of truly buggy code files output by the second step is empty (i.e., the classifier predicts all suspicious code files as non-buggy), the initial recommended list output by the first step is then provided to the developer as the final list.

3.2. Generate the initial buggy file list

As explained in Section 3.1, our technique starts with using existing IR-based bug localization techniques to get an initial buggy code file recommendation list for a given bug. More specifically, we plan to use three representative IR-based bug localization techniques to get the initial list, including BugLocator, AmaLgam, and Blizzard.

- BugLocator [12] — BugLocator uses a revised vector space model (rVSM) to sort all files based on textual similarity (cosine similarity) between bug reports and source code, while also taking into account information about similar bugs that have been fixed previously. It can be taken as a representative of classical IR-based approaches to bug location.
- AmaLgam [14] — AmaLgam is an automated bug localization technique that incorporates various sources of information that may aid in bug localization. Beyond leveraging the textual content of bug reports and source code files, AmaLgam utilizes additional information such as code version history, similar bug reports, and the structural features of both bug reports and source code files. By integrating these diverse data sources, AmaLgam aims to improve the accuracy and effectiveness of localizing buggy files.

- Blizzard [17] — IR-based bug location techniques would perform poorly if a bug report lacks essential information or contains much noise brought by, for example, structured information like stacktrace. Blizzard proposes to reformulate a bug report first and use the bug report with higher quality to perform bug localization. It improves the classical IR-based bug localization with a focus on improving bug report quality.

The three techniques represent different kinds of IRBL techniques. To fully utilize the advantages of three techniques, for a given bug report, we ran them to obtain their suspicious file lists with size = 20 from the code base first. Then, we applied a machine-learning method to merge them and output the 20 files that were most likely to be buggy for the following subset retrieval of truly buggy files.

Specifically, for each file from the three lists, we obtained their ranks and suspicious scores output by three IRBL techniques as instance features (note that since Blizzard only provides file ranks without suspicious scores, each file instance actually would have five features, i.e., three ranks and two suspicious scores), with being buggy or not as their labels. Then, we applied XGBoost [41,42] to the training data to get a classifier. XGBoost is a leading machine-learning method, more specifically, an extension of the gradient boosting algorithm, which is an ensemble learning method that builds a stronger predictive model by combining the predictions of multiple weaker models. It mainly utilizes decision trees as its base models, and these trees are generally shallow, with limited depth and fewer splits, to mitigate the risk of overfitting. It provides parallel tree boosting and could automatically learn to handle missing values within instances (which may happen in our case as Blizzard sometimes would not output a rank list of all code files within a codebase according to its design).

For testing bug reports with also three suspicious lists (aka, 60 recommendation files of three IRBL baselines in total), we used the built XGBoost classifier to predict which one was buggy, sorted the files according to their classification scores, and took the top 20 files with the largest scores as the initial buggy file list for use in the next step of HitMore (i.e., the subset retrieval of truly buggy code files).

3.3. Identify a subset of truly buggy files

In the first step, we could only assure that the initial buggy file list is very likely to contain a truly buggy code file without knowing its file name. This part is to precisely figure out which one from the initial list is truly buggy(or very likely to be truly buggy). We would build a machine learning model to predict whether a code file in the initial list is truly buggy or not. To build the model, we extract three types of instance features shown in Table 1, i.e., features related to bug reports, recommended buggy files, and relations between them, such as textual similarity.

Features related to bug reports. The proposed features related to bug reports include two categories, i.e., bug report quality features and reporter expertise features. Bug report quality is a fundamental factor that largely affects whether, for a new bug report, a bug localization tool can successfully retrieve a buggy code file [17]. Inspired by [43], we consider seven features related to bug report quality, including the existence or not of itemizations, code samples, stack traces, patches, and screenshots, as well as keyword completeness and textual readability of bug reports. We also take into account reporter expertise features. We believe that reporters who submit or fix more bugs are more experienced, then the bug reports submitted by the reporter will be more professional; and if the percentage of bugs submitted by the reporter that are fixed is higher, the bug reports submitted by him/her will be more likely to be valid and of high quality. Hence, for reporter expertise, we mainly calculate the number of bugs fixed by a reporter, the number of bugs fixed by a reporter, and the number of bugs fixed among all submitted bugs by a reporter.

Features related to recommended buggy files. The features related to the suspicious buggy files include code complexity, bug fixing history features, code readability, comments-codes consistency, and involved developer features of code files. More complex code file is more prone to have bugs; we mainly use CK metrics [44], Lines of Code (LoC), and statement numbers to measure the complexity of a code file. As for bug fixing history, for each code file, we mainly consider four features namely bug fixing frequency, bug fixing recency, bug fixing dependencies, and the number of developers who touched the file during bug fixing. According to [15], a frequently fixed source file may be a bug-prone file (bug fixing frequency), and a recently fixed source file may be responsible for newly arrived bugs (bug fixing recency). Bug fix dependencies take into account the dependencies between the current source file and the past buggy files. We believe that if there are dependencies between the current source file and the past buggy files, the current source file is likely to be buggy. We use the java-callgraph suite, which can generate static and dynamic call graphs from the Java system (https://github.corn/gousiosg/java-callgraph), to get the complete call chain of the current source file and count the number of past buggy files that exist in the call chain. The number of developers who touched a code file during bug fixing is also related to the bug-proneness of the code file [47]. If the cumulative number of developers who fix a code file is higher, the more problematic the file may be.

Code readability [48] and code-comment consistency [49] to some extent also reveal the quality of a code file, and further correlate with bug occurrence. A code file with relatively low readability (calculated by using the style tool [43]), or whose code is inconsistent with its comments (measured by topic similarity in this paper) may prevent developers from comprehending the code and hence be more likely to introduce bugs. Inspired by [45,46], we believe that developers of a source code file are also an important factor in bug-introducing. Compared to a developer who focuses on one module, a developer who contributes to multiple modules is more prone to make errors during the development and testing process. In this study, we mainly compute four kinds of developer features including Developer Attention Focus(DAF), Module Attention Focus(MAF) [45], Developer's Structural Scattering and Semantic Scattering [46].

Features related to the relationship between bug reports and recommended buggy files. These features mainly focus on the textual similarity and developer-reporter overlap between bug reports and recommended buggy files.

Considering that code files with greater textual similarity to bug reports are more likely to be associated with the bugs [12], for each bug report, we retrieve the textual similarity between the bug report and these code files in the initial list for the bug. Three types of similarities are calculated, namely raw-text similarity (i.e., Surface Lexical Similarity, API-Enriched Lexical Similarity, Collaborative Filtering Scores, and Class Name Similarity [15]), topic similarity (by capturing higher abstract relevance of two documents in topic level using the Latent Dirichlet Allocation (LDA) model [25]), and word embedding similarity (capturing contextual semantics using word embedding techniques such as Word2Vec that convert both bug reports and source documents into word embedding vectors and calculates their cosine similarity) [12]. All three types of text similarities are calculated based on the lexical terms present in the bug report and the code file. Given that developers (or users) may use different terms to express the same intent [15], we also introduce the WordNet [50] synonym network to extend the lexical terms in bug reports and code files to avoid potential threats.

We also argue that the overlap of developers/reporters involved in bug reports and source code influences the results. For example, if the reporter of a bug report is the same as the developer of the recommended buggy file in the list, the file is more likely related to the bug report. During overlap calculation, we extract reporters and developers from the CCList and Comments of a bug report, as well as those contributing to a code file (through mining Git commit history),

Table 1
Instance features used in identifying a subset of truly buggy files.

Dimension	Feature	Description
Bug report	hasItemizations	Whether the bug report contains itemizations.
	Keyword completeness	The completeness of a bug report by checking whether the keywords appear in a bug report.
	hasCodeSamples	Whether a bug report contains code samples.
	hasStackTraces	Whether a bug report contains stack traces.
	HasPatches	Whether a bug report contains patches.
	hasScreenshots	Whether a bug report contains screenshots.
	Report readability	Following [43], using Kincaid, Automated Readability Index (ARI), Coleman-Liau, RIX, Flesh, Fog, Lix, and SMOG Grade, to measure the readability.
	Report expertise	The number of bugs submitted, fixed by a reporter, and the fixed bug ratio of the reporter.
Recommended buggy file	Bug-fixing recency	The reporting time distance between a bug report and the most recent fixed bug report of a code file.
	Bug-fixing frequency	The number of times a source file has been fixed before the current bug report.
	NDEV	The cumulative number of developers who have changed files in all previous code versions.
	Code complexity	Measured by CK metrics [44], Lines of Code, and Statement number.
	Comments-codes consistency	Consistency between code comments and codes, measured in LDA topic similarity.
	Code readability	Measured in the same as bug report readability.
	DAF(Developer Attention Focus)	Measures how focused the activities are of a developer [45].
	MAF(Module Attention Focus)	Measures how focused the activities are on a module (a code file in this paper) [45].
	Developer's structural scattering	Measures how structurally far the code files modified by a developer are based on the number of subsystems one needs to cross to reach one code file from the other [46].
	Developer's semantic scattering	Measures how much spread the implemented responsibilities of the code files modified by a developer are based on the textual similarity of his/her changed code files [46].
	Bug fix dependencies	The number of past buggy files that have a call/called relationship with the current source file.
Relation between them	Surface lexical similarity	Cosine similarity (using VSM) between a bug report and a code file.
	API-enriched lexical similarity	Cosine similarity (using VSM) between a bug report and a code file extended by its API documentation.
	Collaborative filtering score	Cosine similarity (using VSM) between a given bug report and the bug reports for which a code file was fixed before the bug was reported.
	Class name similarity	The length of source code class name contained in the title of a bug report.
	Topic similarity	Topic similarity (using LDA) between bug reports and source files.
	Semantic similarity	Cosine similarity of word embedding vectors of bug reports and source files.
	Developers and reporters	The number of people both appearing in a bug report and contributing to a code file.

to check how many people are overlapped between them.

Finally, a machine learning model is built to predict truly buggy files using the three types of features extracted from the initial buggy file list.

3.4. Optimize buggy file list

After we obtain the subset of truly (or very likely to be) buggy file(s), our next step is to use them to help retrieve the remaining buggy files. We would particularly focus on three kinds of code files that are related to the given buggy files, i.e., files that have a control flow, data flow, and co-occurrence relationships with those given buggy files. A bug may propagate in code files by following the control flow, data flow. Hence, we believe that code files correlated with a buggy code file through control flow and data flow dependency are more likely to be modified to fix the bug. We say that two code files have a co-occurrence relationship if they are generally changed/committed together or contributed by the same developers. Such a co-occurrence relationship matters in bug localization and bug fixing [15,51]. After we obtained the candidate buggy code files for each relationship, we use a weighted strategy to merge and re-order the code files to generate an optimized buggy file list. This optimized buggy file list is supposed to contain more truly buggy code files than existing state-of-the-art IR-based bug localization techniques.

3.4.1. Code relationship analysis

We mainly focus on three types of code files associated with a given buggy code file based on control flow, data flow, and co-occurrence relationships with the file. For each file from the acquired subset of truly buggy files, a corresponding control flow graph is generated. Based on the analysis, we find the files with control flow dependencies, data flow dependencies, and co-occurrence relationships with the truly buggy file. In other words, three lists of candidate buggy files corresponding to these relationships are generated for each truly buggy file in the subset.

Control flow dependency. It describes the sequential relationship between statements in program execution. It indicates that the execution of a statement depends on the execution result of previous statements. The control dependencies followed in our technique are defined as: let a and b be two nodes of a Control Flow Graph (CFG) of a program; if the execution of b depends on the execution result of a, then b is control-dependent on a.

Data flow dependency. It describes the delivery and dependency of data in a program. In Data Flow Dependencies, the value of a variable is delivered from one program point to another. The definition of data dependencies in our technique is as follows: given a CFG $G = (V, E)$ of a file f, and two nodes n_i, n_j (assuming that j follows i), and $i \neq j$. There exists one path from n_i to n_j in the CFG. If the value calculated in n_i is used in n_j, then n_j data depends on n_i. That is, there is a data dependency between two statements: a variable is defined in one statement, and that variable is used later in another statement.

Co-occurrence dependency. The co-occurrence of two files, to some extent, reveals the relevance between them. In HitMore, we consider three types of co-occurrence dependencies between files: committed together, modified together, and with the same contributor. We first use Git Blame to get the commit time, committer, modified time, and modifier. If two files have the same modification time, they were modified together (i.e., there is a co-occurrence dependency). Similarly, if two files have the same commit time, they also have co-occurrence dependency. Then, we use Git Log to get the developers who have participated in the file's development. If a developer has participated in the development of two files, it is considered that both files have the same contributor. By comparing the time and developers' information, we can identify the code files that have co-occurrence dependencies with each code file in the subset of buggy files.

3.4.2. Weighting strategy for list merge

After code relationship analysis in Section 3.4.1, we would get the following three file lists for the files in the truly buggy file subset:

- List 1 contains files that have control flow dependencies with the files in the subset.
- List 2 contains files that have data flow dependencies with the files in the subset.
- List 3 contains files that have co-occurrence dependencies with the files in the subset.

Our next step is to determine the way to merge these three lists to get a final optimized one. Towards this, we designed a three-step weighting strategy that fully considered the original suspiciousness revealed by mainstream bug localization techniques and the existing dependency relation as follows.

Obtain initial suspicious scores C. For each code file in three lists, we refer to the full recommendation lists of Amalgam, BugLocator, and Blizzard to get its corresponding ranks first; the rank value corresponds to its position in the recommendation list, where a rank of 1 means it is ranked first and is supposed to be the most suspicious. After obtaining the rank of each IRBL technique, we calculate its *ranking* score by inverting its rank (i.e., $3 \rightarrow 1/3$); if a code file is not ranked in a given technique (e.g., not retrieved by Blizzard), its *ranking* score is set as 0. Last, we add the three *ranking* scores and take the sum as the file's initial suspiciousness value. The initial suspicious scores would range from 0 to 3, according to our calculation strategy.

Compute relationship weights W. It is possible that a file may have more than one relationship, e.g., having both control and data-flow dependencies, with a truly buggy code file. In other words, a file from a relation list may also appear in another relation list(s). We think files with more code relationships to a truly buggy code file are likely to be more suspicious than other files. To account for this, for each file in the three relation lists (i.e., the above Lists 1, 2, and 3), we decided to use the number of times it appears across the lists, noted as *occ_num*, as the relationship weight for the file. The weight value would be 1,2, or 3, corresponding to a file appearing in 1 list, 2 lists, or 3 lists, separately.

Reorder based on weighted suspicious score S. After we obtain the initial suspicious score and relationship weight for each file in the three lists, we multiply them and take the weighted suspicious scores as their final scores. Then we sort the files in descending order of their weighted scores to obtain the final optimized buggy file list and return it to developers for further checking.

4. Experiment setup

In this section, we first present the dataset and comparison baselines. Then, we describe the metrics that we used for performance evaluation. Additionally, we provide information regarding the implementation of machine learning algorithms and present some tools that can be used to support our technique.

4.1. Dataset and baselines

In this paper, six open-source Java projects are selected for experiments: Tomcat, AspectJ, Lucene, ZooKeeper, OpenJPA, and Hibernate-ORM. These projects are widely used for static bug localization techniques. They are from different domains and are of varied code scales. We totally collected 7609 bug reports from the six projects. Among them, Tomcat and AspectJ are from the dataset shared by [15]. For the remaining four projects, we crawled their code and bug reports up to November 2018 and linked the bug reports with their corresponding buggy code files through heuristic rules obtained by manually analyzing the commit logs.

Table 2
Basic statistics of six experimental projects.

Project	Time period	Bug reports	Lines of code	Number of bugs with multiple buggy files (Ratio)
ZooKeeper	2008-06-09–2018-11-12	470	140,175	305 (64.89%)
OpenJPA	2006-08-11–2018-11-16	533	723,284	311 (58.35%)
Tomcat	2002-07-06–2014-01-18	992	573,208	339 (34.17%)
AspectJ	2002-03-13–2014-01-10	563	690,560	376 (66.79%)
Hibernate-ORM	2004-08-22–2018-11-16	1285	1,068,498	805 (62.65%)
Lucene	2007-07-12–2018-11-14	1454	1,762,563	1079 (74.21%)

Through manual analysis of bug reports and commit logs, we found that developers tended to add projectName-bugID in their commit logs to tell others which bugs they fixed. For example, by adding "ZOOKEEPER-3217" to say they fixed a ZooKeeper bug with ID 3217. Hence, we used the heuristic rule projectName-bugID to obtain bug reports which could potentially be linked with their buggy code files. For those bugs which failed to match the heuristic rule, we further searched their raw bugIDs through project commit logs, e.g., using "3217" rather than "ZOOKEEPER-3217". After the two-step searching, we would get a list of linked bug report candidates. To filter out some false positive candidates, we manually checked each candidate and obtained 6278 linked bug reports from the four projects.

Following [15], we further removed bug reports that linked to multiple commits or shared the same commit with others, as it was not clear which files were relevant. Some bug reports with no deleted or modified code files were also ignored as it was not applicable to predict buggy files which were not created yet in the buggy version of project code. After we linked bug reports with bug-fixing commits, following the strategy used in [15], for each bug report, we checked out the version right before the bug-fixing commit, and took the deleted and modified code files as the buggy files that contained the bug (adding files were ignored as they did not even exist yet when the bug report was initially reported). Table 2 shows the basic information about the six experimental projects, including the project name, the time period of bug reports, the number of bug reports, and the number of code lines.

To evaluate the effectiveness of combining three localization techniques to produce an initial list of buggy files, we use AmaLgam, BugLocator and Blizzard as baselines. In evaluating HitMore, we also use the three techniques as comparison baselines. These techniques selected for comparison are well-performed representatives of static, especially IR-based bug localization techniques.

4.2. Metrics

HitMore mainly includes two parts, one is to retrieve a subset of truly buggy code files from an initial buggy file recommendation list. The other part is to retrieve the remaining buggy files based on the subset through three kinds of code relationship analysis. The result combination of the two parts constitutes the output of HitMore to developers. Correspondingly, we used two kinds of performance metrics to evaluate the effectiveness of our technique, namely binary classification metrics and bug localization metrics. The binary classification metrics are used to evaluate how effective we are in retrieving the subset of truly buggy files. We take it as a binary classification problem that identifies truly buggy files from an initial buggy file recommendation list. Hence, we use Accuracy, Precision, Recall, and F1-score, which are commonly used evaluation metrics for binary classification models, to measure the performance of the identification step for a subset.

The bug localization metrics are used to measure the final overall performance of locating buggy files of a bug localization technique. As our HitMore is also an information retrieval-based technique that provides a buggy file recommendation list for a bug report (as a query), we also adopt the commonly used Accuracy@K, MAP, and MRR for localization performance evaluation. Further, we also designed two new metrics HitCount@N and multiCompleteness@All, to better evaluate our HitMore in locating multiple-buggy-file bugs. The definitions of Accuracy@K, MAP, MRR, HitCount@N, and multiCompleteness@All are as follows.

Accuracy@K. Accuracy@K is defined as the ratio of bugs that has at least one truly buggy code file successfully retrieved in a recommendation of size K.

MAP (mean average precision). This metric returns the mean value of the average detection rate of all bug reports. The average detection rate is an average over different recall rate points for a search result, and is defined as follows:

$$P(j) = \frac{\text{number of positive instances in top } j}{j} \tag{1}$$

$$\text{Avg } P_i = \cdot \sum_{j=1}^{N} \frac{P(j) \times pos(j)}{\text{number of positive instances}} \tag{2}$$

$$MAP = \frac{1}{Q} \sum_{i=1}^{Q} \text{Avg } P_i \tag{3}$$

where $pos(j)$ denotes whether the instance ranked j is relevant to the bug report: relevant is 1, otherwise 0. $P(j)$ denotes the precision of the retrieval at position j. Avg P_i denotes the average retrieval precision of query i. MAP denotes the mean of the average precision for Q queries.

MRR (mean reciprocal rank). This metric returns the mean of the inverse rank of a series of queries. The inverse rank of a query refers to the inverse of 1st relevant document's rank, i.e., the inverse of the rank of the buggy program module, which is defined as follows:

$$MRR = \frac{1}{Q} \sum_{i=1}^{Q} \frac{1}{rank_i} \tag{4}$$

where $rank_i$ denotes the rank of the 1st relevant file.

HitCount@N. HitCount@N is defined as the number of bugs that have N truly buggy files successfully retrieved in a buggy file recommendation list of a certain size K.

MultiCompleteness@All. It measures the retrieval completeness for multiple-buggy-code-file bugs. As shown in the following formula, multiCompleteness@All is defined as the ratio of bugs that have all truly buggy files retrieved in a buggy file recommendation list of size K.

$$multiCompleteness@All = \frac{q}{Q} \tag{5}$$

where q denotes the number of bugs for which all multiple (>1) truly buggy files are successfully retrieved in the recommendation list of buggy files. Q is the number of bugs associated with multiple (>1) truly buggy code files.

4.3. Implementation and tool supports

In identifying a subset of truly buggy files, we divide the dataset into training and test datasets. We train the model on the training dataset and validate the model with the test dataset. Five cross-validations were used. In this paper, five machine learning algorithms are tested, i.e., Random Forest (RF), Support Vector Machine (SVM), Decision Tree (DT), Naive Bayes (NB), and Logistic Regression (LR). During the model construction process, we also used the random oversampling strategy to deal with the class imbalance problem (one class having more instances than the other).

In obtaining the code files that have control/data flow and co-occurrence dependency with the subset of buggy files, we used the available tools and commands: (i) For the control flow analysis of code files, the Understand API tool was used first. For each code file f in the subset of buggy files, generate its control flow graph control_flow_graph = java_file.ents("Control Flow Graph")[0]. Each node of the control flow graph is then read to find the nodes that simultaneously satisfy the criteria of being reachable from f and having at least two direct successors. These nodes (files) have a control flow relationship with the code file f. (ii)For the data flow analysis of code files, we follow the control flow graph obtained in (i). If the value obtained in node n_i is used in node n_j, then n_j data depends on n_i. (iii) For analyzing the co-currency dependency of code files, we first use the Git blame command to obtain the commit time, commit person, change time, and person information of each code file. Then, we match the time and the person's name to get the code files that have co-occurrence dependencies with the truly buggy files in the subset. The replication package of our study is available at https://github.com/LyraXv/HitMore.

5. Experiment results

This section presents the results of our experiment by analyzing and answering the following two research questions.

RQ1: Can we effectively retrieve the contained truly buggy code file(s) from the initial recommendation list?

RQ2: What is the overall performance of HitMore in locating bugs especially those bugs with multiple buggy code files?

5.1. RQ1: Can we effectively retrieve the contained truly buggy file(s) from the initial recommendation list?

The successful retrieval of truly buggy code files from the initial recommendation list (i.e., the subset of truly buggy files) is the basis for subsequent retrieval of remaining truly buggy files. Knowing the subset retrieval performance can help us better understand our HitMore and reveal potential improvement directions of our technique. Before checking that, we first checked whether the initial recommendation list indeed contains truly buggy code files, as it is the data basis for subset retrieval. Through analyzing the obtained initial lists of size = 20, we find that 94.58%, 80.86%, 89.21%, 74.96%, 81.17% and 92.98% of them contain at least one truly buggy code file for the bug reports of six projects (i.e., ZooKeeper, OpenJPA, Tomcat, AspectJ, Hibernate-ORM, and Lucene). We think these ratios are acceptable in our following subset retrieval experiments.

As we transform the subset identification problem into a binary classification one, we present the overall retrieval performance in terms of Accuracy, Precision, Recall, and F1 scores. The importance of instance features used to build classification models is also analyzed. Detailed results are as follows.

Subset Retrieval Performance. As mentioned in Section 3.3, we build a binary classification model based on three kinds of bug/code file features to predict whether a code file from the initial recommendation list is truly buggy or not. Since we have no idea which classifier is best suited for this task, we test five typical machine learning (ML) models, including Decision Tree (DT), Support Vector Machine (SVM), Naive Bayes (NB), Logistic Regression (LR), and Random Forest (RF). Furthermore, considering that an ML algorithm generally has some tunable hyperparameters, which practitioners often choose to optimize for better performance in real application scenarios, we also performed hyperparameter tuning on the five ML algorithms. With reference to existing studies [52–55], we identified the hyperparameters to be tuned for each ML method and their respective parameter ranges (shown in Table A.10 in the Appendix Section). We then conducted experiments by systematically combining all possible values of the hyperparameters for every ML method. Finally, the hyperparameter settings that delivered the best performance (regarding F1 scores) were selected to train

Table 3
Subset retrieval performance of truly buggy files.

Project	Classifier	Accuracy	Precision	Recall	F1-Score
ZooKeeper	DT	81.5%	86.0%	81.5%	83.4%
	SVM	73.6%	**87.2%**	73.6%	78.4%
	NB	54.7%	86.1%	54.7%	63.3%
	LR	72.4%	87.1%	72.4%	77.5%
	RF	**85.5%**	86.7%	**85.5%**	**86.1%**
OpenJPA	DT	85.3%	89.4%	85.3%	87.2%
	SVM	73.5%	90.8%	73.5%	79.9%
	NB	42.6%	90.3%	42.6%	53.3%
	LR	73.1%	**90.9%**	73.1%	79.6%
	RF	**88.6%**	89.9%	**88.6%**	**89.2%**
Tomcat	DT	88.1%	90.4%	88.1%	89.1%
	SVM	80.3%	91.3%	80.3%	84.5%
	NB	47.7%	90.3%	47.7%	58.4%
	LR	78.5%	91.3%	78.5%	83.3%
	RF	**92.7%**	**91.7%**	**92.7%**	**92.1%**
AspectJ	DT	85.5%	90.5%	85.5%	87.7%
	SVM	87.0%	90.3%	87.0%	88.5%
	NB	31.4%	90.3%	31.4%	41.1%
	LR	72.0%	**91.4%**	72.0%	79.1%
	RF	**91.6%**	91.0%	**91.6%**	**91.3%**
Hibernate-ORM	DT	88.5%	90.5%	88.5%	89.4%
	SVM	67.5%	90.7%	67.5%	75.7%
	NB	69.5%	91.1%	69.5%	77.0%
	LR	76.1%	**91.7%**	76.1%	81.8%
	RF	**92.6%**	91.5%	**92.6%**	**91.9%**
Lucene	DT	82.5%	85.8%	82.5%	83.9%
	SVM	75.5%	87.3%	75.5%	79.6%
	NB	73.5%	86.0%	73.5%	77.9%
	LR	75.3%	87.4%	75.3%	79.5%
	RF	**88.9%**	**87.7%**	**88.9%**	**88.2%**

the model used in the subset retrieval.

Table 3 shows the classification performance of five ML methods after hyperparameter tuning on six projects (the concrete hyperparameter settings that lead to the best performance for each ML method could be found in Table A.11 in the Appendix Section).

From Table 3, we can find that in terms of weighted Accuracy, Precision, Recall, and F1-Score, the Random Forest classifier performs best among the five classifiers and demonstrates excellent results on the Tomcat, AspectJ and Hibernate-orm project sets (the Accuracy, Precision, Recall, and F1-score are as high as 91.0% to 92.7%). The Naive Bayes performs most poorly in these projects, with F1-scores ranging from 41.1% to 77.9% in six experimental projects. With Random Forest, we can obtain a best F1-score that ranges from 86.1% to 92.1% across six experimental projects. These results indicate that our built classification models are able to help us figure out the truly buggy code files contained in the initial recommendation list. This lays a good foundation for us to retrieve other remaining buggy code files in our next step.

Feature Importance. As shown in Table 1, three kinds of features are used to build the classifier. We perform two kinds of analysis to understand the importance of these features: (i) How effective is the model built on a single kind of features? (ii) Which exact features are more influential in predicting truly buggy files? During model building, we chose the Random Forest classifier as it was found to perform best in overall classification performance (in Table 3). The details are as follows.

For (i), we first built three prediction models on each kind of features separately. Then, we compare their performance with the one built on all features in terms of F1-Score, a metric that manages the balance between accurately identifying buggy files (recall) and reducing false positives (precision), which is crucial in bug localization tasks. Table 4 shows the F1-Score results. From the table, we can find that, as expected, the model built on all features achieved much better performance than the ones built on every single kind of features. This

Table 4
F1-score of the Random Forest Models Based on Individual Feature Subsets.

Project	Dimension	F1-score
ZooKeeper	**All**	**86.1%**
	Bug Reports	64.8%
	Recommended Buggy Files	81.3%
	Reltionship Between Them	77.9%
OpenJPA	**All**	**89.2%**
	Bug Reports	63.1%
	Recommended Buggy Files	86.3%
	Reltionship Between Them	81.6%
Tomcat	**All**	**92.1%**
	Bug Reports	62.3%
	Recommended Buggy Files	87.9%
	Reltionship Between Them	85.1%
AspectJ	**All**	**91.3%**
	Bug Reports	62.8%
	Recommended Buggy Files	89.9%
	Reltionship Between Them	80.5%
Hibernate-ORM	**All**	**91.9%**
	Bug Reports	67.6%
	Recommended Buggy Files	89.4%
	Reltionship Between Them	86.2%
Lucene	**All**	**88.2%**
	Bug Reports	61.3%
	Recommended Buggy Files	83.7%
	Reltionship Between Them	80.3%

means all three kinds of features are needed in the subset retrieval of truly buggy files. As for the three kinds of features themselves, the features related to recommended buggy files outperform the other two feature groups by about 2.8% to 27.1% in F1-Score. The results of the relationship feature group are relatively lower but not by much than that of the buggy file feature group. Models building on the bug report feature group perform worst, obtaining an F1 score ranging from 61.3% to 67.6% on six projects.

For (ii), we followed the process of [56,57] to analyze the importance of individual features. Specifically, we first conduct correlation analysis by using the R package Hmisc to reduce the covariance between features [17,32]. Then, we perform redundancy analysis by using the redun function to remove redundant features. After that, we build random forest models on left features and apply statistical tests to identify important features. More detailedly, we first build a random forest model using the R package randomForest, with 10-fold cross-validation applied. During model training, the importance function in the randomForest package is used to compute the importance of a factor based on the OOB (the internal error estimate of a random forest classifier). Based on the importance values of individual features from prediction models, the Scott-Knott ESD test [58] is applied to determine the most important features. Further, to better measure how effective these features are, we use the Wilcoxon rank sum test [35] along with the Bonferroni correction [59] to compare the statistical significance of differences between code files with and without bugs. The differences are measured by Cliff's delta effect size [60].

Table 5 shows the top 5 features that are most indicative in predicting truly buggy files for each project. The last column, δ, shows Cliff's delta effect size and corresponding difference magnitude; for effect size values less than 0.147, between 0.147 and 0.33, between 0.33 and 0.474, and above 0.474, map them to "Negligible", "Small", "Medium", and "Large" difference magnitude, respectively [61]. The significance levels are also marked with stars in the table, from which we can tell that the associated statistical tests all show statistical significance at the p-value < 0.05, more specifically < 0.001.

Among these six projects, nine features belong to the top 5 most important features, i.e., Topic Similarity, Class Name Similarity, Surface Lexical Similarity, Collaborative Filtering Score, Semantic Similarity, DAF, MAF, Developers Semantic Scattering, cf.RIX(code file readability). Further, as shown in Table 5, two features, namely Topic Similarity

Table 5
Top 5 most important features and their Cliff's delta effect size in predicting truly buggy files.

Project	Feature name	δ
ZooKeeper	Surface lexical similarity	0.319 (small)***
	Collaborative filter score	−0.272 (small)***
	Topic similarity	0.234 (small)***
	Class name similarity	−0.152 (small)***
	Semantic similarity	−0.112 (negligible)***
OpenJPA	Collaborative filtering score	0.257 (small)***
	Surface lexical similarity	0,229 (small)***
	Topic similarity	0.161 (small)***
	Class name similarity	0.124 (negligible)***
	Semantic similarity	−0.113 (negligible)***
Tomcat	Surface lexical similarity	0.366 (medium)***
	Class name similarity	0.280 (small)***
	Topic similarity	0.213 (small)***
	Collaborative filter score	0.213 (small)***
	Semantic similarity	−0.109 (negligible)***
AspectJ	Collaborative filter score	0.259 (small)***
	MAF	−0.109 (negligible)***
	Topic similarity	0.105 (negligible)***
	DAF	−0.101 (negligible)***
	Class name similarity	0.092 (negligible)***
Hibernate-ORM	Surface lexical similarity	0.288 (small)***
	Class name similarity	0.233 (small)***
	cf.RIX(Readability)	0.232 (small)***
	Topic similarity	0.227 (small)***
	DAF	−0.199 (small)***
Lucene	Surface lexical similarity	0.294 (small)***
	Class name similarity	0.257 (small)***
	Topic similarity	0.215 (small)***
	Semantic similarity	−0.184 (small)***
	Developers semantic scattering	0.038 (negligible)***

Significance levels: * $p < 0.05$, ** $p < 0.01$, *** $p < 0.001$.

and Class Name Similarity, appear in all six projects, indicating their predictive value for identifying truly buggy files across diverse datasets.

Regarding their effect size, we can find that all features show a negligible to medium difference magnitude between buggy and non-buggy code files. The feature Surface Lexical Similarity (which appeared as the top 5 most important feature in five projects) shows a relatively larger effect size compared to other features, with one medium and four small effect sizes, indicating stronger discriminatory power for distinguishing between truly buggy files and non-buggy files. Topic Similarity presents five small and one negligible effect size; Class Name Similarity shows four small and two negligible effect sizes; Collaborative Filtering Score appeared four times as the top 5 most important feature among six projects, with all effect sizes also being small. The effect sizes of different features in Table 5 also, to some extent, indicate that the reported classification performance in this study is achieved through the combined contributions of all features.

5.2. RQ2: What is the overall performance of HitMore in locating bugs especially those bugs with multiple buggy code files?

HitMore is a technique mainly designed to improve locating bugs with multiple buggy code files. However, as we cannot know in advance whether a newly arrived bug is associated with single or multiple buggy code files or not in practice, it then becomes important to make sure that HitMore would not perform worse than other bug localization techniques in general locating cases. Hence, in evaluating HitMore, we first present its overall locating performance by using general locating metrics (Accuracy@K, MAP, and MRR) like other locating techniques. Then, we particularly check to what extent our HitMore can improve multiple-buggy-code-file bugs compared to existing techniques. For each comparison of our HitMore and three baselines, we further conduct statistical analysis by running the Wilcoxon Rank Sum test and using Cliff's Delta effect size, to check whether our observed performance differences between them have statistical significance or not, and if so, how large the differences are. Result details are as follows.

Overall Locating Performance. Table 6 shows the overall locating performance of HitMore and three strong baselines in terms of Accuracy@K (i.e., A@K in the table), MAP, and MRR. From the table, we can find that in general bug locating cases (without distinguishing single or multiple-buggy-code-file bugs or not), HitMore did not lose to comparison baselines. Instead, in all cases, HitMore achieved better results than baselines. Specifically, compared to three baselines, HitMore improved the MAP by 0.09–0.27, 0.06–0.19 and 0.08–0.15, respectively, and improved the MRR by 0.07–0.3, 0.03–0.18 and 0.07–0.13, respectively, across six projects. By referring to the definition of MAP and MRR, the corresponding improvements of HitMore indicate that HitMore could place more truly buggy code files into the top-K recommendation list (higher MAP), and make developers check fewer files to find the first truly buggy one (higher MRR). The largest improvements of MAP (0.13–0.27) and MRR (0.1–0.3) over three baselines happened on the Lucene project. Considering that Lucene has 74.21% multiple-buggy-code-file bugs, such a quite substantial improvement over baselines indirectly indicates our HitMore performs much better in locating multiple-buggy-code-file bugs than existing techniques. As for the Accuracy@K metric, we can find that in all cases, HitMore obtains the best Accuracy@K, which also releases a positive signal of our HitMore in locating general bugs in real development situations. For example, HitMore could improve Accuracy@20 by 3.19%–27.94%, 1.07%–16.5% and 3.41%–15.45% over three baselines across six projects, respectively.

By referring to the statistical analysis results in Table 8, we can conclude that the observed performance differences between HitMore and three baselines are all statistically significant at the significance level of p-value < 0.05. The magnitude of these differences are medium to large according to the Cliff's Detla effect size. These results further support the effectiveness of our HitMore in locating bugs.

Extraction Effectiveness of Multiple-Buggy-Code-File Bugs. The results in Table 6 reveal that HitMore could achieve better performance than existing bug-locating techniques in general cases. Our next step is to figure out to what extent our HitMore outperforms existing techniques in handling multiple-buggy-code-file bugs. This is mainly measured by our metrics HitCount@N (the number of bugs that have N truly buggy code files retrieved in a recommendation list) and multiCompleteness@All (the ratio of bugs having all multiple truly buggy code files retrieved in the recommendation list). Table 7 shows the corresponding HitCount@N and Completeness@All results for HitMore and three comparison baselines. For better comparison, we provide the number of multiple-buggy-code-file bugs and their corresponding ratios in the Project column. we also add a delta number in the () beside the HitCount@N values in the table. The delta number in the () is obtained by minus HitCount@N of a technique from that of HitMore. For example, the 200 (−58) in the All column (HitCount@All) of AmaLgam, means that AmaLgam could retrieve all truly buggy code files for 200 bugs, with 58 bugs fewer than HitMore which retrieves all truly buggy code files for 258 bugs. We consider three values of N: One, Two, and All.

From Table 7, we can find that in terms of all three HitCount@N (HitCount@One, HitCount@Two, HitCount@All), HitMore outperformed three baselines in all six experimental projects except one (In OpenJPA, the HitCount@Two of HitMore is 7 fewer than that of AmaLgam). This means HitMore is able to improve existing IR-based bug-locating techniques in different extraction levels, either retrieving only one or multiple truly buggy code files. For example, the values of HitCount@One (the number of bugs having at least one truly buggy file retrieved) are 17-359, 5-212 and 16-153 higher than that of AmaLgam, BugLocator and Blizzard across six projects, respectively; the corresponding delta values of HitCount@Two/HitCount@All are

Table 6
Overall locating performance of HitMore and three baselines in terms of accuracy@K, MAP, and MRR.

Project	Approach	A@1	A@5	A@10	A@20	MAP	MRR
ZooKeeper	AmaLgam	34.26%	66.38%	73.62%	82.55%	0.39	0.48
	BugLocator	52.55%	78.94%	87.45%	91.91%	0.51	0.65
	Blizzard	50.00%	73.83%	82.13%	89.57%	0.49	0.61
	HitMore	**56.81%**	**81.70%**	**89.57%**	**92.98%**	**0.57**	**0.68**
OpenJPA	AmaLgam	30.39%	57.60%	68.28%	77.86%	0.36	0.43
	BugLocator	28.52%	56.29%	65.85%	75.61%	0.35	0.41
	Blizzard	26.83%	50.28%	60.60%	69.61%	0.31	0.38
	HitMore	**38.46%**	**64.54%**	**72.80%**	**81.05%**	**0.45**	**0.50**
Tomcat	AmaLgam	28.73%	52.12%	60.99%	67.54%	0.35	0.39
	BugLocator	45.16%	69.56%	79.64%	86.19%	0.51	0.57
	Blizzard	41.53%	67.35%	76.62%	83.67%	0.48	0.53
	HitMore	**51.11%**	**74.80%**	**81.65%**	**87.70%**	**0.58**	**0.61**
AspectJ	AmaLgam	19.01%	36.94%	48.85%	59.68%	0.22	0.29
	BugLocator	20.07%	43.34%	55.77%	68.38%	0.22	0.32
	Blizzard	20.07%	39.08%	48.49%	58.44%	0.21	0.30
	HitMore	**31.08%**	**55.42%**	**64.83%**	**74.07%**	**0.35**	**0.43**
Hibernate-ORM	AmaLgam	17.51%	35.41%	43.35%	51.75%	0.21	0.26
	BugLocator	23.81%	46.07%	56.26%	63.19%	0.25	0.33
	Blizzard	29.73%	50.43%	59.92%	67.78%	0.31	0.40
	HitMore	**39.92%**	**64.05%**	**71.91%**	**79.69%**	**0.44**	**0.51**
Lucene	AmaLgam	29.30%	57.22%	67.19%	75.45%	0.30	0.42
	BugLocator	50.07%	76.27%	83.70%	88.51%	0.41	0.62
	Blizzard	49.11%	76.89%	84.32%	88.93%	0.44	0.62
	HitMore	**62.65%**	**85.01%**	**89.68%**	**92.85%**	**0.57**	**0.72**

Table 7
Extraction effectiveness of multiple buggy code files in terms of HitCount@N and multiCompleteness@All (mC).

Project	Approach	HitCount@N(Δ)			mC(%)
		One	Two	All	
ZooKeeper (305,64.89%)	Amalgam	388(−49)	164(−47)	200(−58)	24.92
	BugLocator	432(−5)	201(−10)	236(−22)	29.18
	Blizzard	421(−16)	195(−16)	225(−33)	26.89
	HitMore	**437**	**211**	**258**	**35.08**
OpenJPA (311,58.35%)	Amalgam	415(−17)	**120(+7)**	230(−11)	**16.72**
	BugLocator	403(−29)	101(−12)	229(−12)	15.43
	Blizzard	371(−61)	80(−33)	195(−46)	10.29
	HitMore	**432**	113	**241**	16.40
Tomcat (339,34.17%)	Amalgam	670(−200)	128(−50)	499(−191)	20.35
	BugLocator	855(−15)	163(−15)	658(−32)	23.60
	Blizzard	830(−40)	151(−27)	638(−52)	21.53
	HitMore	**870**	**178**	**690**	**28.32**
AspectJ (376,66.79%)	Amalgam	336(−81)	107(−36)	125(−40)	6.91
	BugLocator	385(−32)	100(−43)	149(−16)	6.38
	Blizzard	329(−88)	65(−78)	131(−34)	2.93
	HitMore	**417**	**143**	**165**	**8.51**
Hibernate-ORM (805,62.65%)	Amalgam	665(−359)	171(−129)	287(−211)	7.33
	BugLocator	812(−212)	220(−80)	395(−103)	9.57
	Blizzard	871(−153)	205(−95)	395(−103)	7.58
	HitMore	**1024**	**300**	**498**	**13.54**
Lucene (1079,74.21%)	Amalgam	1097(−253)	584(−225)	511(−265)	26.14
	BugLocator	1286(−64)	581(−228)	537(−239)	22.15
	Blizzard	1293(−57)	691(−118)	628(−148)	29.66
	HitMore	**1350**	**809**	**776**	**41.52**

-7-225/11-265, 10-228/12-239 and 16-118/33-148, separately.

Further, by referring to the multiCompleteness@All values in the table(i.e., the mC column), we can find that our HitMore outperformed three baselines in the locating performance of mutiple-buggy-code-file bugs over all six projects (except on the OpenJPA, where AmaLgam obtained slightly better mC over HitMore). In six projects, HitMore is able to retrieve all buggy code files for 8.51% to 41.52% multiple-buggy-code-file bugs; while for three baselines, their corresponding values are 6.91%–26.14%, 6.38%–29.18% and 2.93%–29.66%, respectively. The absolute improvement in muliCompleteness@All between HitMore and three baselines (i.e., AmaLgam, BugLocator, and Blizzard) are up to 15.38% (6.83% on average), 19.36% (6.18% on average), and 11.86% (7.42% on average), respectively.

The obtained statistical test results and Cliff's Delta effect size in Table 9, indicate that the performance differences between HitMore and three baselines in locating multiple-buggy-code-file bugs all present statistical significance at the p-value < 0.05. These results indicate that HitMore indeed performs better than baselines. In terms of HitCount@N (N: One, Two, and All), the difference magnitude between HitMore and three baselines ranges from small to medium according to the Cliff's Delta effect size. While in the multiCompleteness@All (mC) metric, HitMore shows two medium difference magnitudes (over AmaLgam and Blizzard) among three baselines (over BugLocator, it is small). The overall larger difference magnitude (based on effect size) between HitMore and three baselines in the multiCompleteness@All metric further demonstrates the effectiveness of our HitMore in locating multiple-buggy-code-file bugs over existing techniques; as according to the definition of multiCompleteness@All, it serves as a more pure metric in measuring locating performance for the particular kind of multiple-buggy-code-file bugs, compared to HitCount@N.

6. Threats to validity

In this Section, we present the threats to validity of our study. These threats can be categorized into external validity, internal validity, construct validity and conclusion validity.

External validity. In this study, all experiments and analyses are conducted on six open-source software (OSS) projects written in Java. We cannot guarantee that the arrived conclusions could be applicable to other OSS or industry projects written in Java or other languages. However, these projects are all well-known and widely-used projects in practice; they come from different domains and are of different sizes. This, to some extent, reveals the effectiveness of our HitMore used in real development scenarios, especially on those projects with a number of multiple-buggy-code-file bugs. Further replicate studies on more projects are encouraged to generalize our conclusions.

Internal validity. During performance comparison with baselines, we directly used the reported configurations of those baselines (with the best performance in the original papers) in the paper. As our evaluations were conducted on different projects, it is possible that the

Table 8
Cliff's delta effect size and Wilcoxon rank-sum test results in terms of accuracy@K, MAP, and MRR.

Technique pair	Acc@1	Acc@5	Acc@10	Acc@20	MAP	MRR
HitMore vs. Amalgam	0.944(large)*	0.722(large)*	0.722(large)*	0.722(large)*	0.917(large)*	0.917(large)*
HitMore vs. BugLocator	0.444(medium)*	0.333(medium)*	0.333(medium)*	0.333(medium)*	0.611(large)*	0.389(medium)*
HitMore vs. Blizzrd	0.500(large)*	0.444(medium)*	0.389(medium)*	0.389(medium)*	0.611(large)*	0.417(medium)*

Significance levels: * $p < 0.05$, ** $p < 0.01$, *** $p < 0.001$.

Table 9
Cliff's delta effect size and Wilcoxon rank-sum test results in terms of HitCount@N and multiCompleteness@All (mC).

Technique Pair	HitCount@N(Δ)			mC(%)
	One	Two	All	
HitMore vs. Amalgam	0.389(medium)*	0.389(medium)*	0.278(small)*	0.333(medium)*
HitMore vs. BugLocator	0.306(small)*	0.222(small)*	0.278(small)*	0.278(small)*
HitMore vs. Blizzrd	0.278(small)*	0.278(small)*	0.278(small)*	0.389(medium)*

Significance levels: * $p < 0.05$, ** $p < 0.01$, *** $p < 0.001$.

configurations may not best suit our projects. We made such a decision mainly based on two considerations. Firstly, all parameters were dedicatedly tuned with various projects in the original papers; this, to a large extent, guaranteed the applicability of individual parameters. Secondly and more importantly, from the adoption of tools in real practice, it is not likely for a software project to conduct parameter tunes, especially when they have limited data or when the parameter tune is complex or time-consuming. Instead, they are very likely to directly use the recommend/default configurations when checking whether a tool is useful to their project or not. Thus, we believe that it is practical and reasonable to use the default configurations of baselines to compare their performance with HitMore.

Construct validity. Threats to construct validity relate to the suitability of our evaluation metrics on multiple-buggy-file bugs. Although traditional metrics such as MAP are valuable for assessing how well buggy files are ranked in a recommendation list, they fall short in evaluating whether all buggy files associated with a single bug are effectively located. This limitation underscores the need for more targeted metrics in scenarios involving multiple buggy files. To address this issue, we introduced two metrics: HitCount@N and multiCompleteness@All. HitCount@N evaluates the number of bugs for which the technique successfully identifies N buggy files within the recommendation list, while multiCompleteness@All measures the technique's ability to completely locate all buggy files for a given bug. While these metrics aim to better reflect the challenges of locating multiple buggy files and align with developers' needs for efficiency and completeness, we remain cautious about their limitations and view them as an ongoing attempt to refine the evaluation process for multiple-buggy-code-file bugs.

Conclusion validity. The ability to draw conclusions could be affected by various factors. To avoid HitMore obtaining good results by chance and overfitting, we conducted a five-fold cross validation and compared the performance of locating results using standard metrics, widely used in evaluating IRBL techniques [4]. As a further analysis, we performed statistical tests to determine the statistical significance of the results and used Cliff's delta effect size to quantify the difference between HitMore and baselines statistically.

7. Conclusion

In this paper, we propose a technique, HitMore, which aims to improve the locating performance of those bugs associated with multiple buggy code files. The basic idea of HitMore is to generate an initial recommendation list that is rather likely to contain at least a truly buggy code file, then use a prediction model based on bug reports and code semantic features to retrieve a subset of truly buggy code files from the initial list, last, fully leverage the code relations between the subset and the codebase to retrieve the remaining truly buggy code files. The experimental results on six projects demonstrate that our HitMore could significantly improve the locating performance for those bugs with multiple buggy code files without decreasing but improving the locating performance in general bug locating scenarios.

CRediT authorship contribution statement

Hui Xu: Writing – review & editing, Writing – original draft, Formal analysis, Validation, Software, Visualization. **Zhaodan Wang:** Writing – review & editing, Writing – original draft, Methodology, Investiagation, Data Curation, Software. **Weiqin Zou:** Writing – review & editing, Supervision.

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.

Acknowledgments

This work is supported by the National Natural Science Foundation of China (No. 62002161), 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.

Appendix

A.1. Hyperparameters tuning for five classifiers

Table A.10 presents the hyperparameters to be tuned for each machine learning method and the value ranges of these hyperparameters during the tuning process. Table A.11 presents the concrete configurations of the hyperparameters that lead to best classification performance in terms of F1 scores during tuning process for each machine learning classifier over six experimental projects.

Table A.10
Hyperparameters and corresponding value range during tunning process for five classifiers.

Classifier	Hyperparameters	Default	Value range
Decision tree	min_samples_split	2	[1,60]
	min_samples_leaf	1	[1,60]
SVM	kernel	'rbf'	['linear', 'rbf' , 'poly', 'sigmoid']
	cost	1	$2^x, x \in [-10, 10]$
	gamma	1/n_features	$2^x, x \in [-10, 10]$
	degree	3	[2,5]
Naive bayes	var_smoothing	1e−9	[0.0,1.0]
Logistic regression	C	1	[1,10]
Random forest	max_features	sqrt(n_features)	[0.1,1.0], $\log_2 n\_features$
	max_samples	None	[0.1,1.0]

Table A.11
Best hyperparameter configurations for each classifier over six experimental projects.

Classifier	Project	Parameters
Random forest	ZooKeeper	'max_features': 0.5, 'max_samples': 0.1
	openJPA	'max_features': 0.4, 'max_samples': 0.1
	Tomcat	'max_features': 0.2, 'max_samples': 0.4
	AspectJ	'max_features': 0.2, 'max_samples': 0.7
	Hibernate-ORM	'max_features': 0.7, 'max_samples': 0.3
	Lucene	'max_features': 0.8, 'max_samples': 0.1
SVM	ZooKeeper	'C': 4, 'gamma': 0.015625, 'kernel':'rbf'
	openJPA	'C': 128, 'gamma': 0.001953125, 'kernel': 'rbf'
	Tomcat	'C': 1, 'gamma': 0.03125, 'kernel': 'rbf'
	AspectJ	'C': 1024, 'gamma': 0.0625, 'kernel': 'rbf'
	Hibernate-ORM	'C': 64, 'kernel': 'linear'
	Lucene	'C': 256, 'gamma': 0.001953125, 'kernel': 'rbf'
Naive bayes	ZooKeeper	'var_smoothing': 0.1
	openJPA	'var_smoothing': 0.2
	Tomcat	'var_smoothing': 0.1
	AspectJ	'var_smoothing': 1e−09
	Hibernate-ORM	'var_smoothing': 0.1
	Lucene	'var_smoothing': 0.1
Logistic Regression	ZooKeeper	'C': 7
	openJPA	'C': 1
	Tomcat	'C': 1
	AspectJ	'C': 1
	Hibernate-ORM	'C': 1
	Lucene	'C': 1
Decision Tree	ZooKeeper	'min_samples_leaf': 12, 'min_samples_split': 53
	openJPA	'min_samples_leaf': 13, 'min_samples_split': 18
	Tomcat	'min_samples_leaf': 12, 'min_samples_split': 16
	AspectJ	'min_samples_leaf': 30, 'min_samples_split': 27
	Hibernate-ORM	'min_samples_leaf': 13, 'min_samples_split': 23
	Lucene	'min_samples_leaf': 41, 'min_samples_split': 60

A.2. Bug localization with time-ordered data

In the main text, to reduce variance in evaluation results and avoid overfitting to a single split, we performed random five-fold cross-validation in subset retrieval of truly buggy code files. The subsequent locating performance of HitMore is also computed based on all bug data collected from recommendation results corresponding to individual folds. Given that in practice, bug reports generally appear in chronological order, to better understand the robustness of our HitMore, we complemented the experiments of using the first 80% of bug reports as training data and the last 20% of bug reports as testing data, with all bug reports being ordered in their reporting time from oldest to newest.

Table A.12 shows the overall locating performance of HitMore and three baselines over the 20% testing bug reports. As shown in Table A.12, HitMore still outperformed three baselines in most cases. The Accuracy@K advantage of HitMore is more obvious than three baselines when K is smaller. This means HitMore can place more truly buggy code files closer to the top of the recommendation lists than existing techniques. This is also reflected by the improved MAP (by 0.03–0.19, 0.07–0.22 and 0.1–0.18, respectively) and MRR (by 0.0–0.21, 0.03–0.16, 0.06–0.17, respectively).

Table A.13 further presents the HitCount@N and multiCompleteness@All results of HitMore and three baselines. From the table, we can observe that in terms of HitCount@N (N: One, Two, and All), HitMore outperforms the three comparison baseline methods in all cases except

Table A.12
Overall locating performance of HitMore and three baselines on latest 20% bug reports in terms of Accuracy@K, MAP, and MRR.

Project	Approach	A@1	A@5	A@10	A@20	MAP	MRR
ZooKeeper	AmaLgam	50.53%	75.79%	81.05%	87.37%	0.51	0.61
	BugLocator	60.00%	78.95%	85.26%	89.47%	0.53	0.69
	Blizzard	51.58%	71.58%	81.05%	85.26%	0.49	0.62
	HitMore	**65.26%**	**85.26%**	**90.53%**	**93.68%**	**0.62**	**0.74**
OpenJPA	AmaLgam	29.91%	**56.07%**	**62.62%**	71.03%	0.35	**0.42**
	BugLocator	27.10%	47.66%	54.21%	65.42%	0.29	0.37
	Blizzard	26.17%	44.86%	54.21%	65.42%	0.27	0.36
	HitMore	**31.78%**	53.27%	59.81%	**71.03%**	**0.38**	**0.42**
Tomcat	AmaLgam	30.50%	54.00%	64.00%	70.50%	0.35	0.41
	BugLocator	42.50%	68.00%	73.50%	**83.50%**	0.46	0.54
	Blizzard	36.00%	65.00%	74.00%	82.50%	0.43	0.49
	HitMore	**46.00%**	**72.50%**	**77.00%**	**83.50%**	**0.53**	**0.57**
AspectJ	AmaLgam	23.89%	41.59%	55.75%	61.95%	0.29	0.34
	BugLocator	27.43%	45.13%	55.75%	67.26%	0.31	0.36
	Blizzard	23.89%	44.25%	52.21%	57.52%	0.28	0.33
	HitMore	**32.74%**	**51.33%**	**62.83%**	**69.91%**	**0.40**	**0.43**
Hibernate-ORM	AmaLgam	24.71%	43.92%	51.37%	60.00%	0.28	0.34
	BugLocator	24.71%	47.84%	60.39%	73.73%	0.27	0.35
	Blizzard	23.92%	45.49%	54.51%	63.14%	0.28	0.34
	HitMore	**41.18%**	**61.57%**	**71.37%**	**80.39%**	**0.46**	**0.51**
Lucene	AmaLgam	42.12%	65.75%	74.32%	83.56%	0.39	0.53
	BugLocator	45.89%	73.29%	82.53%	86.99%	0.36	0.58
	Blizzard	47.26%	78.42%	85.62%	89.04%	0.44	0.62
	HitMore	**65.07%**	**86.99%**	**90.07%**	**93.84%**	**0.58**	**0.74**

Table A.13
Extraction effectiveness of multiple buggy code files on latest 20% bug reports in terms of HitCount@N and multiCompleteness@All (mC).

Project	Approach	HitCount@N(Δ)			mC(%)
		One	Two	All	
ZooKeeper (60,63.16%)	Amalgam	83	38	53	36.67%
	BugLocator	85	38	46	28.33%
	Blizzard	81	40	48	33.33%
	HitMore	**89**	**45**	**59**	**43.33%**
OpenJPA (63,58.88%)	Amalgam	**76**	**20**	**44**	**12.70%**
	BugLocator	70	14	38	7.94%
	Blizzard	70	12	37	6.35%
	HitMore	**76**	19	**44**	11.11%
Tomcat (81,40.50%)	Amalgam	141	29	92	22.22%
	BugLocator	**167**	34	115	18.52%
	Blizzard	165	29	114	14.81%
	HitMore	**167**	**35**	**119**	**24.69%**
AspectJ (40,35.40%)	Amalgam	70	10	46	5.00%
	BugLocator	76	8	54	5.00%
	Blizzard	65	7	47	2.50%
	HitMore	**79**	**11**	**56**	**7.50%**
Hibernate-ORM (175,68.63%)	Amalgam	153	45	59	**9.71%**
	BugLocator	188	34	69	4.57%
	Blizzard	161	32	67	6.29%
	HitMore	**205**	**53**	**85**	9.14%
Lucene (215,73.63%)	Amalgam	244	132	129	34.88%
	BugLocator	254	108	105	20.93%
	Blizzard	260	135	121	26.51%
	HitMore	**274**	**164**	**159**	**41.86%**

the comparison with AmaLgam on OpenJPA in HitCount@Two. The multiCompleteness@All shows that HitMore generally performs better than baselines except in two cases where involving the comparisons with AmaLgam (HitMore located completely one multiple-buggy-code-file bug fewer than AmaLgam on OpenJPA and Hibernate-ORM, hence leading to a slightly smaller multiCompleteness@All).

The results from both five-fold cross-validation (reported in the main text) and the time-ordered 80-20 strategies support the conclusion that HitMore outperforms baseline approaches in bug localization. Meanwhile, for multiple-buggy-code-file bugs, the advantage of HitMore under the time-ordered 80-20 split is relatively smaller compared to the five-fold cross-validation results presented in the main text. This could suggest that newer bug reports may exhibit distinct characteristics compared to older ones, which our current approach has not fully captured. Additionally, since the time-ordered 80-20 split experiment only allows for a single experiment for each project, the potential variability of single-run results could also contribute to this discrepancy. In the future, larger-scale datasets – including more recent bug reports for testing – and a wider variety of projects could provide a more comprehensive evaluation and further validate the effectiveness of our HitMore in practice.

Data availability

The replication package of our study is available at https://github.com/LyraXv/HitMore.

References

[1] Weiqin Zou, David Lo, Zhenyu Chen, Xin Xia, Yang Feng, Baowen Xu, How practitioners perceive automated bug report management techniques, IEEE Trans. Softw. Eng. 46 (8) (2018) 836–862.

[2] Daming Zou, Jingjing Liang, Yingfei Xiong, Michael D Ernst, Lu Zhang, An empirical study of fault localization families and their combinations, IEEE Trans. Softw. Eng. 47 (2) (2019) 332–347.

[3] Xiaoyuan Xie, Tsong Yueh Chen, Fei-Ching Kuo, Baowen Xu, A theoretical analysis of the risk evaluation formulas for spectrum-based fault localization, ACM Trans. Softw. Eng. Methodol. (TOSEM) 22 (4) (2013) 1–40.

[4] W Eric Wong, Ruizhi Gao, Yihao Li, Rui Abreu, Franz Wotawa, A survey on software fault localization, IEEE Trans. Softw. Eng. 42 (8) (2016) 707–740.

[5] Plinio S Leitao-Junior, Diogo M Freitas, Silvia R Vergilio, Celso G Camilo-Junior, Rachel Harrison, Search-based fault localisation: A systematic mapping study, Inf. Softw. Technol. 123 (2020) 106295.

[6] Sungmin Kang, Gabin An, Shin Yoo, A preliminary evaluation of llm-based fault localization, 2023, arXiv preprint arXiv:2308.05487.

[7] W Eric Wong, Ruizhi Gao, Yihao Li, Rui Abreu, Franz Wotawa, Dongcheng Li, Software fault localization: An overview of research, techniques, and tools, in: Handbook of Software Fault Localization: Foundations and Advances, Wiley Online Library, 2023, pp. 1–117.

[8] Xin Xia, David Lo, Information retrieval-based techniques for software fault localization, in: Handbook of Software Fault Localization: Foundations and Advances, Wiley Online Library, 2023, pp. 365–391.

[9] Partha Chakraborty, Mahmoud Alfadel, Meiyappan Nagappan, RLocator: Reinforcement learning for bug localization, IEEE Trans. Softw. Eng. (2024).

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

[11] Ripon K Saha, Matthew Lease, Sarfraz Khurshid, Dewayne E Perry, Improving bug localization using structured information retrieval, in: 2013 28th IEEE/ACM International Conference on Automated Software Engineering, ASE, IEEE, 2013, pp. 345–355.

[12] Jian Zhou, Hongyu Zhang, David Lo, 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, 2012, pp. 14–24.

[13] Chu-Pan Wong, Yingfei Xiong, Hongyu Zhang, Dan Hao, Lu Zhang, Hong Mei, Boosting bug-report-oriented fault localization with segmentation and stack-trace analysis, in: 2014 IEEE International Conference on Software Maintenance and Evolution, IEEE, 2014, pp. 181–190.

[14] Shaowei Wang, David Lo, Version history, similar report, and structure: Putting them together for improved bug localization, in: Proceedings of the 22nd International Conference on Program Comprehension, 2014, pp. 53–63.

[15] Xin Ye, Razvan Bunescu, Chang 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.

[16] Zhaoqiang Guo, Huicong Zhou, Shiran Liu, Yanhui Li, Lin Chen, Yuming Zhou, Baowen Xu, Information retrieval based bug localization: Research problem, progress, and challenges, J. Softw. 31 (9) (2020) 2826–2854.

[17] Mohammad Masudur Rahman, Chanchal K. Roy, Improving ir-based bug localization with context-aware query reformulation, in: Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering, 2018, pp. 621–632.

[18] Jeongju Sohn, Shin Yoo, Fluccs: Using code and change metrics to improve fault localization, in: Proceedings of the 26th ACM SIGSOFT International Symposium on Software Testing and Analysis, 2017, pp. 273–283.

[19] Zhengliang Li, Zhiwei Jiang, Xiang Chen, Kaibo Cao, Qing Gu, Laprob: a label propagation-based software bug localization method, Inf. Softw. Technol. 130 (2021) 106410.

[20] Fan Fang, John Wu, Yanyan Li, Xin Ye, Wajdi Aljedaani, Mohamed Wiem Mkaouer, On the classification of bug reports to improve bug localization, Soft Comput. 25 (2021) 7307–7323.

[21] Jifeng Xuan, Martin Monperrus, Learning to combine multiple ranking metrics for fault localization, in: 2014 IEEE International Conference on Software Maintenance and Evolution, IEEE, 2014, pp. 191–200.

[22] Dylan Callaghan, Bernd Fischer, Improving spectrum-based localization of multiple faults by iterative test suite reduction, in: Proceedings of the 32nd ACM SIGSOFT International Symposium on Software Testing and Analysis, 2023, pp. 1445–1457.

[23] Xi Xiao, Renjie Xiao, Qing Li, Jianhui Lv, Shunyan Cui, Qixu Liu, BugRadar: Bug localization by knowledge graph link prediction, Inf. Softw. Technol. 162 (2023) 107274.

[24] Shivani Rao, Avinash Kak, 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, 2011, pp. 43–52.

[25] David M. Blei, Andrew Y. Ng, Michael I. Jordan, Latent dirichlet allocation, J. Mach. Learn. Res. 3 (Jan) (2003) 993–1022.

[26] Stacy K. Lukins, Nicholas A. Kraft, Letha H. Etzkorn, Source code retrieval for bug localization using latent dirichlet allocation, in: 2008 15Th Working Conference on Reverse Engineering, IEEE, 2008, pp. 155–164.

[27] Andrian Marcus, Andrey Sergeyev, Vaclav Rajlich, Jonathan I Maletic, An information retrieval approach to concept location in source code, in: 11th Working Conference on Reverse Engineering, IEEE, 2004, pp. 214–223.

[28] Scott Deerwester, Susan T Dumais, George W Furnas, Thomas K Landauer, Richard Harshman, Indexing by latent semantic analysis, J. Am. Soc. Inf. Sci. 41 (6) (1990) 391–407.

[29] Anh Tuan Nguyen, Tung Thanh Nguyen, Jafar Al-Kofahi, Hung Viet Nguyen, Tien N Nguyen, A topic-based approach for narrowing the search space of buggy files from a bug report, in: 2011 26th IEEE/ACM International Conference on Automated Software Engineering, ASE 2011, IEEE, 2011, pp. 263–272.

[30] Bunyamin Sisman, Avinash C. Kak, Incorporating version histories in information retrieval based bug localization, in: 2012 9th IEEE Working Conference on Mining Software Repositories, MSR, IEEE, 2012, pp. 50–59.

[31] Xin Ye, Hui Shen, Xiao Ma, Razvan Bunescu, Chang Liu, From word embeddings to document similarities for improved information retrieval in software engineering, in: Proceedings of the 38th International Conference on Software Engineering, 2016, pp. 404–415.

[32] Bunyamin Sisman, Avinash C. Kak, Assisting code search with automatic query reformulation for bug localization, in: 2013 10th Working Conference on Mining Software Repositories, MSR, IEEE, 2013, pp. 309–318.

[33] Oscar Chaparro, Juan Manuel Florez, Andrian Marcus, Using observed behavior to reformulate queries during text retrieval-based bug localization, in: 2017 IEEE International Conference on Software Maintenance and Evolution, ICSME, IEEE, 2017, pp. 376–387.

[34] Misoo Kim, Eunseok Lee, A novel approach to automatic query reformulation for ir-based bug localization, in: Proceedings of the 34th ACM/SIGAPP Symposium on Applied Computing, 2019, pp. 1752–1759.

[35] An Ngoc Lam, Anh Tuan Nguyen, Hoan Anh Nguyen, Tien N Nguyen, Bug localization with combination of deep learning and information retrieval, in: 2017 IEEE/ACM 25th International Conference on Program Comprehension, ICPC, IEEE, 2017, pp. 218–229.

[36] Xiangxin Meng, Xu Wang, Hongyu Zhang, Hailong Sun, Xudong Liu, Improving fault localization and program repair with deep semantic features and transferred knowledge, in: Proceedings of the 44th International Conference on Software Engineering, 2022, pp. 1169–1180.

[37] Leila Yousofvand, Seyfollah Soleimani, Vahid Rafe, Automatic bug localization using a combination of deep learning and model transformation through node classification, Softw. Qual. J. (2023) 1–19.

[38] Anil Koyuncu, Tegawendé F. Bissyandé, Dongsun Kim, Kui Liu, Jacques Klein, Martin Monperrus, Yves Le Traon, D&c: A divide-and-conquer approach to ir-based bug localization, 2019, arXiv preprint arXiv:1902.02703.

[39] Saket Khatiwada, Miroslav Tushev, Anas Mahmoud, On combining ir methods to improve bug localization, in: Proceedings of the 28th International Conference on Program Comprehension, 2020, pp. 252–262.

[40] Tien-Duy B. Le, Ferdian Thung, David Lo, Will this localization tool be effective for this bug? Mitigating the impact of unreliability of information retrieval based bug localization tools, Empir. Softw. Eng. 22 (2017) 2237–2279.

[41] Tianqi Chen, Carlos Guestrin, Xgboost: A scalable tree boosting system, in: Proceedings of the 22nd Acm Sigkdd International Conference on Knowledge Discovery and Data Mining, 2016, pp. 785–794.

[42] Zhi-Hua Zhou, Ensemble Methods: Foundations and Algorithms, CRC Press, 2012.

[43] Thomas Zimmermann, Rahul Premraj, Nicolas Bettenburg, Sascha Just, Adrian Schroter, Cathrin Weiss, What makes a good bug report? IEEE Trans. Softw. Eng. 36 (5) (2010) 618–643.

[44] Shyam R. Chidamber, Chris F. Kemerer, A metrics suite for object oriented design, IEEE Trans. Softw. Eng. 20 (6) (1994) 476–493.

[45] Daryl Posnett, Raissa D'Souza, Premkumar Devanbu, Vladimir Filkov, Dual ecological measures of focus in software development, in: 2013 35th International Conference on Software Engineering, ICSE, IEEE, 2013, pp. 452–461.

[46] Dario Di Nucci, Fabio Palomba, Giuseppe De Rosa, Gabriele Bavota, Rocco Oliveto, Andrea De Lucia, A developer centered bug prediction model, IEEE Trans. Softw. Eng. 44 (1) (2017) 5–24.

[47] Thomas J. Ostrand, Elaine J. Weyuker, Robert M. Bell, Programmer-based fault prediction, in: Proceedings of the 6th International Conference on Predictive Models in Software Engineering, 2010, pp. 1–10.

[48] Raymond P.L. Buse, Westley R. Weimer, Learning a metric for code readability, IEEE Trans. Softw. Eng. 36 (4) (2009) 546–558.

[49] Darryl Jarman, Jeffrey Berry, Riley Smith, Ferdian Thung, David Lo, Legion: Massively composing rankers for improved bug localization at adobe, IEEE Trans. Softw. Eng. 48 (8) (2021) 3010–3024.

[50] George A. Miller, WordNet: a lexical database for English, Commun. ACM 38 (11) (1995) 39–41.

[51] Emerson Murphy-Hill, Thomas Zimmermann, Christian Bird, Nachiappan Nagappan, The design of bug fixes, in: 2013 35th International Conference on Software Engineering, ICSE, IEEE, 2013, pp. 332–341.

[52] Philipp Probst, Anne-Laure Boulesteix, Bernd Bischl, Tunability: Importance of hyperparameters of machine learning algorithms, J. Mach. Learn. Res. 20 (53) (2019) 1–32.

[53] Rafael Gomes Mantovani, Tomáš Horváth, André LD Rossi, Ricardo Cerri, Sylvio Barbon Junior, Joaquin Vanschoren, André CPLF de Carvalho, Better trees: an empirical study on hyperparameter tuning of classification decision tree induction algorithms, Data Min. Knowl. Discov. (2024) 1–53.

[54] Rui Shu, Tianpei Xia, Jianfeng Chen, Laurie Williams, Tim Menzies, How to better distinguish security bug reports (using dual hyperparameter optimization), Empir. Softw. Eng. 26 (2021) 1–37.

[55] Bingting Chen, Weiqin Zou, Biyu Cai, Qianshuang Meng, Wenjie Liu, Piji Li, Lin Chen, An empirical study on the potential of word embedding techniques in bug report management tasks, Empir. Softw. Eng. 29 (5) (2024) 122.

[56] Lingfeng Bao, Zhenchang Xing, Xin Xia, David Lo, Shanping Li, Who will leave the company?: a large-scale industry study of developer turnover by mining monthly work report, in: 2017 IEEE/ACM 14th International Conference on Mining Software Repositories, MSR, IEEE, 2017, pp. 170–181.

[57] Yuan Tian, Meiyappan Nagappan, David Lo, Ahmed E. Hassan, What are the characteristics of high-rated apps? a case study on free android applications, in: 2015 IEEE International Conference on Software Maintenance and Evolution, ICSME, IEEE, 2015, pp. 301–310.

[58] Chakkrit Tantithamthavorn, Shane McIntosh, Ahmed E Hassan, Kenichi Matsumoto, An empirical comparison of model validation techniques for defect prediction models, IEEE Trans. Softw. Eng. 43 (1) (2016) 1–18.

[59] David H. Wolpert, William G. Macready, An efficient method to estimate bagging's generalization error, Mach. Learn. 35 (1999) 41–55.

[60] Norman Cliff, Dominance statistics: Ordinal analyses to answer ordinal questions, Psychol. Bull. 114 (3) (1993) 494.

[61] Norman Cliff, Ordinal Methods for Behavioral Data Analysis, Psychology Press, 2014.