DigBug – Pre/Post-processing Operator Selection for Accurate Bug Localization

Kisub Kim[a], Sankalp Ghatpande[b], Kui Liu[c], Anil Koyuncu[e], Dongsun Kim[d,*], Tegawendé F. Bissyandé[b], Jacques Klein[b] and Yves Le Traon[b]

[a]Singapore Management University, Singapore
[b]Interdisciplinary Centre for Security, Reliability and Trust (SnT), University of Luxembourg, Luxembourg
[c]The Software Engineering Application Technology Lab at Huawei, China
[d]School of Computer Science and Engineering, Kyungpook National University, South Korea
[e]Faculty of Engineering and Natural Sciences, Sabanci University, Turkey

ARTICLE INFO

Keywords:
Bug report, Bug localization, Fault Localization, Bug Characteristics, Information Retrieval, Operator Combination

ABSTRACT

Bug localization is a recurrent maintenance task in software development. It aims at identifying relevant code locations (e.g., code files) that must be inspected to fix bugs. When such bugs are reported by users, the localization process become often overwhelming as it is mostly a manual task due to incomplete and informal information (written in natural languages) available in bug reports. The research community has then invested in automated approaches, notably using *Information Retrieval* techniques. Unfortunately, reported performance in the literature is still limited for practical usage. Our key observation, after empirically investigating a large dataset of bug reports as well as workflow and results of state-of-the-art approaches, is that most approaches attempt localization for every bug report without considering the different characteristics of the bug reports. We propose DigBug as a straightforward approach to specialized bug localization. This approach selects pre/post-processing operators based on the attributes of bug reports; and the bug localization model is parameterized in accordance as well. Our experiments confirm that departing from "one-size-fits-all" approaches, DigBug outperforms the state-of-the-art techniques by 6 and 14 percentage points, respectively in terms of MAP and MRR on average.

1. Introduction

Bugs are prevalent in software development processes. Extensive testing and code reviews help detect and address many of these before the software system is released. However, many bugs persist in the code even after the software has been shipped. These bugs are later discovered by end-users who report them to the development team. During the entire lifetime of a project, large numbers of bug reports may overwhelm the available resources of the development team. For example, Apache Hive [1] project has recorded more than 22k bug reports in its 12 years of existence.

Users fill in bug reports and, in most cases, are written in natural language. These reports may be provided by internal or even external developers; and more often authored by software users. A bug report describes an occurrence of unexpected behavior of the software. The report may contain additional artifacts such as stack traces or logs that provide information about a crash within the software. Upon receiving a bug report, an effort must be undertaken to localize the relevant snippet of code that leads to unexpected behavior. In general, bug localization based on textual reports focuses on identifying the relevant buggy file within the code repository.

*Corresponding author.
kisubkim@smu.edu.sg (K. Kim); contact@sghatpande.eu (S. Ghatpande); brucekuiliu@gmail.com (K. Liu); anil.koyuncu@sabanciuniv.edu (A. Koyuncu); darkrsw@knu.ac.kr (D. Kim); tegawende.bissyande@uni.lu (T.F. Bissyandé); jacques.klein@uni.lu (J. Klein); Yves.LeTraon@uni.lu (Y.L. Traon)
ORCID(s):

However, this process of bug localization is inherently complex due to the large and ever-growing code repository combined with the time taken to perform this process manually. Automating bug localization is thus viewed as an important endeavor for the cost and time efficiency of the development and maintenance process [2].

The software engineering community have attempted to address the bug localization problem, mainly by viewing it as an information retrieval (IR) problem: a bug report is viewed as an input query while the collection of source code files within the code repository is treated as the search space; relevant files are then matched, and a ranked list of suspicious files is presented to the development teams. Several different IR methods [3, 4, 5] have been leveraged for this task by state-of-the-art approaches [3, 4, 6, 7, 8, 9, 10, 11, 12, 13].

In the last decade, many approaches have been proposed to improve the application of IR techniques. Generally, these approaches attempt to leverage additional information such as similarity to bug reports associated with revision history [10], considering more fine-grained attributes (e.g., the distinction between method names [14], stack traces tokens [15], and the natural language tokens [16]), different algorithms (e.g., vector space model [17] and latent semantic indexing [4]).

Unfortunately, as concluded in a recent study by Lee et al. [18], the performance of the state-of-the-art research is still rather limited. Generally, for any given project, the state-of-the-art approach provides results that are below 70% in terms of Mean Average Precision (MAP) and Mean Reciprocal Rank (MRR). Recent attempts to improve these results include the approach of 'Divide & Conquer' by Koyuncu et

al. [8] that investigated the impact of dividing bug reports into groups to improve the performance of localization. Another study by Panichella et al. [19] showed the impact on performance when considering the removal of special characters, identifier splitting.

We study the importance of the pre/post-processing operators (e.g., tokenization, stopword removal, stemming, and presence of code entities) that must be applied explicitly to different bug reports to achieve the best localization performance with a standard IR method. Applying different configurations of bug localization techniques has been studied by Thomas et al. [20]. Their study, however, still focuses on the 'one-configuration-fits-all' strategy rather than considering attributes of different reports and applying different configurations depending on the different attributes. Similarly, the study by Binkley et al. [21] focused more on the impact of choosing different query configurations to improve the performance.

In this work, we propose DIGBUG as an IR-based bug localization technique, which learns to apply specific pre/post-processing operators to different groups of bug reports based on their characteristics. Our hypothesis here is that different operators are effective depending on the attributes of a bug report. These attributes include '*having an attachment?*', '*written by a developer or a user*', '*containing a stack trace*'. Our preliminary study (Section 3) partially confirms the hypothesis. Based on the result of the preliminary study, we design DIGBUG, which applies different combinations of pre /post-processing operators to each bug report according to its attributes.

We evaluate the performance of DIGBUG against the subjects from Bench4BL [18], the most significant available benchmark for bug localization. As a result, DIGBUG achieves a better performance if the combination of pre/post-processing operators is selected specifically for the attributes and characteristics of the subject. Overall, building on a simple classical VSM-based approach, the evaluation results show that DIGBUG outperforms the state-of-the-art IRBL techniques up to 6 and 14 percentage points in MAP and MRR, respectively.

The contributions of our approach can be summarized as follows:

- We motivate the study by investigating the role of pre/post-processing operators and identify the combination of operators that provide better performance for different subjects. This already implies that we have to consider different operators for different characteristics and motivate the main approach's hypothesis.
- We build a tool named DIGBUG, an IR-based bug localization technique that selects a best-performing pre/post-processing operator combination based on the characteristics of incoming bug reports. The operators are applied before (i.e., pre-processing), and after (i.e., post-processing) any IR technique performs localization.
- We evaluate the performance of DIGBUG on one [18] of the most significant benchmark designed for bug localization. The evaluation measures MAP and MRR against the state-of-the-art bug localization benchmark. The results show that DIGBUG can locate bugs with high accuracy and outperform the existing state-of-the-art techniques.

The remainder of the paper is organized as follows. Section 2 presents background details on IR-based bug localization such as pre/post-processing operators and performance metrics. In Section 3, we conduct a preliminary study on investigating the varying impact of the common operators in bug localization that forms the RQ1 in this study. The details of our approach are introduced in Section 4. We raise further research questions (i.e., RQ2 and RQ3), evaluate the tool, and discuss its results in Section 5. After discussing related work in Section 6, we conclude our work in Section 7.

2. Background & Motivation

2.1. Bug Localization & IRBL

Bug localization is a software maintenance task that pinpoints the location of suspicious code within the program that is potentially responsible for defects or issues. In many cases, the information about these defects and issues comes in bug reports created by users and developers who have discovered them. These bug reports comprehensively describe the context of a particular bug within the software that users and developers discover. The bug reports consist of textual sentences and paragraphs; instead, they also consist of different metadata and attributes, including code-related entities, log files detailing the issues, and information about affection version of the software. An example of a bug report can be seen in Figure 1 that consists of (1) summary (the title of the report), (2) metadata (e.g., type and priority), and (3) description. It specifies attributes such as code entity (e.g., *GenericMessage*, *invokeListener*), reporter (on the right-hand side), and stack trace in the description.

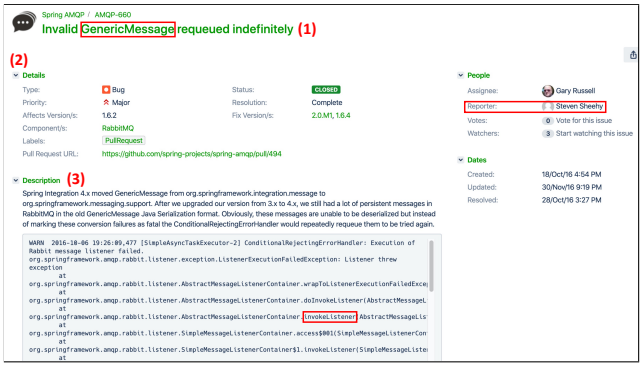


Figure 1: Example of bug reports excerpted from `https://jira.spring.io/browse/AMQP-660`.

On receipt of such bug report, there are two lines of techniques available to localize the bug: spectrum-based fault localization (SBFL) [22, 23, 24, 25] and information retrieval based bug localization (IRBL) [26, 27, 28, 7, 6, 29]. The former line of techniques can be applied only when test cases

associated with the bug are given as they use coverage information (i.e., spectra). Based on the information, SBFL techniques provide a ranked list of suspicious locations (generally, specific lines or blocks in a program). IRBL, on the other hand, takes textual information from bug reports to figure out suspicious locations (often files). These approaches leverage techniques derived from natural language processing such as Latent Dirichlet Allocation (e.g., [30]), Vector Space Model (e.g., [31]), Latent Semantic Analysis (e.g., [32]), and Clustering (e.g., [33]).

Generally, many bug reports are submitted without any test case [34], due to which IRBL techniques are widely applied for bug localization. In addition, it is lightweight and scalable, and it relies mainly on static information such as bug reports and source code. One of the key intuitions behind IRBL is the existence of common tokens in bug reports and source code. For example, users who submit a bug report often mention a specific functionality or an error message. This information is in the form of tokens that are likely to be matched with tokens found in source code files. Many bug reports contain stack traces (e.g., the trace mentioned in the example bug report) and code entities (e.g., method and class names like *invokeListener* in the example bug report) that increase the accuracy in detecting the location of defects. This task is also regarded as identifying feature locations [35], in which the input query is a set of tokens from a bug report, the data source is the text (tokens) in source code files, and the output is a subset (or ranked list) of source code files in a subject.

2.2. Pre/Post-processing operators for IRBL

Bug localization with information retrieval techniques requires the raw input data (source code files and bug reports) to be pre-processed that includes reducing noise, extracting attributes, and improving the resulting model. Therefore, IRBL approaches need to apply relevant pre-processing operators on the inputs [20]. Similarly, some post-processing operators (e.g., filters for irrelevant test files and re-order the ranked list of suspicious files with specific information) may be leveraged to adjust further the outputs yielded by the IR ranking system.

In this work, we focus on the pre/post-processing operators that are listed in Table 1 as they are commonly adopted in recent IRBL studies. In our work, we apply the default pre-processing (i.e., `preBasic`, the basic pre-processing operator for tokenization commonly used for natural language processing (NLP) tasks [36, 37]) that is applied by all the techniques in the literature. Along with the basic operator, stop word removal (`SWR`) and stemming (`STM`) are widely chosen by several IRBL techniques [38, 39, 40, 41, 42]. Dropping out special characters (`SPC`) and splitting on camel case (`CMC`)[1] are common operators to process, in particular, the text in programs. For our study, we consider a single post-processing operator (`CE`), which prioritizes source files whose name appears as code entities (such as Custom variable name, Method name, Class name) in the bug reports. This operator is implemented following the insights highlighted in recent bug localization approaches [27, 6, 29]. After the retrieval process, `CE` of the bug report can be checked/matched from the *Code Entity* database. We designed it assuming we store all the source code tokens already in practice. It relies on the weights upon the number of entities found in the report. The higher the number of detected code entities, the lower is the weight assigned to it in the ranking process.

[1]Note: Underscore splitting is similar to this operator but we address `CMC` in this work since experiment subjects are written in Java.

Table 1
Pre/Post-processing operators used in this study.

Pre-processing operators	
Name	Description
preBasic	Tokenizing words by white space, tab, and new line characters.
SWR	Removing the stop words (e.g., "I" and "She").
STM	Stemming to find the root of each words (e.g., "consultant" and "consulting" → "consult").
SPC	Cleaning up special characters and keywords (e.g., "/" → "" (empty string)).
CMC	Splitting camel cases (e.g., "JavaClass" → "Java Class").
Post-processing operator	
CE	Extracting code entities (e.g., Custom variable name, Method name, Class name) from the bug reports or related attachments after parsing it, then emphasising a high rank on this, if one exists.

2.3. Performance Metrics

Bug localization techniques are generally evaluated by comparing the localization estimations against ground truth data, inferred by considering details from the fix patched developed for a set of resolved, and thus closed bug reports: files that are impacted by these patches are considered as bug locations. The assessment of localization performance is usually conducted by considering MAP and MRR as they are two representative performance metrics for most IR-based bug localization approaches (e.g., [43, 18, 27]).

- **Precision**: More accurately referred to as *Precision@k*, is the metric that represents an estimation of how many files are correctly recommended within the given top k files. It is expressed as follows:

$$\mathrm{P}(k) = \frac{\text{\# of buggy files in top k}}{k} \tag{1}$$

- **Average Precision (AP)**: This aggregates precision values of several positively recommended files for a single bug report. The average precision of a given report is computed by:

$$\mathrm{AP} = \sum_{i=1}^{N} \frac{\mathrm{P}(i) \cdot pos(i)}{\text{\# of positive instances}} \tag{2}$$

where N is the number of ranked files by a given IRBL technique, i is a rank in the list of recommended files. $pos(i)$ indicates whether the i-th file in the ranked list is a buggy file (i.e., $pos(i) \in \{0, 1\}$). For example, AP= 0.5

represents that an IRBL technique can make correct recommendations with 50% of probability within top k recommendations.

- **Mean Average Precision (MAP)**: The MAP is computed by taking the mean value of AP for a set of bug reports rather than a single one:

$$\text{MAP} = \frac{1}{M}\sum_{j=1}^{M}\text{AP}(j) \tag{3}$$

where M is the number of given reports. AP(j) is the average precision of the j-th report. If MAP=p, at least one file is likely to be a correct recommendation for every $\frac{1}{p}$ file in the ranked list.

- **Mean Reciprocal Rank (MRR)**: This computes the mean value of the position of the first buggy file in the ranked list given by an IRBL technique, following this equation:

$$\text{MRR} = \frac{1}{M}\sum_{i=1}^{M}\frac{1}{f\text{-rank}_i} \tag{4}$$

where M is the number of all bug reports and f-rank$_i$ means the position of the first buggy file in the ranked list for the i-th bug report. For example, assuming MRR= 0.5, an IRBL technique can locate at least one correct file to fix within top two recommendations (i.e., $\frac{1}{\text{MRR}} = \frac{1}{0.5} = 2$).

3. Preliminary Study

We hypothesize that **all pre/post-processing operators are neither necessary nor effective on all inputs (i.e., bug reports and source code), and thus should not be uniformly applied by IRBL techniques**. However, in most of the existing IRBL studies, all the pre/post-processing operators selected in the study are applied blindly. We claim that each input from a different project (subject) may have specific characteristics, and the results of applying each pre/post-processing operator could have varying impacts on the performance of IRBL.

We motivate the need for a selective approach by investigating the varying impact of the common operators listed in Section 2.2. The key indicator of impact is the performance of a straightforward approach to bug localization. Therefore, the objective of this preliminary study is to show how different combinations of the operators affect the results of IRBL. To conduct this study, we sample a subset of subjects from a bug localization benchmark (cf. Section 3.1 for details) and apply all possible combinations of operators with a baseline IR technique (cf. Section 3.2 for details).

3.1. Subjects

For this preliminary study, we select four mid-sized (in terms of the number of bug reports) projects, MATH, SHDP, LDAP, and SWS, from the Bench4BL [18] benchmark. Bench4BL is a comprehensive, reproducible package of six state-of-the-art techniques for bug localization that were executed on a large dataset of 9,459 bug reports collected from 46 Java projects. The four projects are selected since they have enough bug reports but not too many to show the impact of operator combinations on the bug localization performance. We use the same benchmark to evaluate our approach as shown in Section 5. The full list of subjects of Bench4BL is listed in Table 2. It should be noted that for our experiment we only consider the initial information from the bug report i.e. we do not consider additional information and attachment provided after the bug has been reported.

Table 2
Subjects used in our study, Bench4BL [18].

Group	Subject	# Source files (Max)	# Major versions	# Bug reports
Apache	CAMEL	14,522	60	1,469
	HBASE	2,714	70	836
	HIVE	4,651	21	1,241
	CODEC	115	9	42
	COLLECTIONS	525	7	92
	COMPRESS	254	15	113
	CONFIGURATION	447	11	133
	CRYPTO	82	1	8
	CSV	29	3	14
	IO	227	13	91
	LANG	305	16	217
	MATH	1,617	15	245
	WEAVER	113	1	2
JBoss	ENTESB	252	3	47
	JBMETA	858	5	26
	ELY	68	3	25
	SWARM	727	6	58
	WFARQ	126	1	1
	WFCORE	3,598	16	361
	WFLY	8,990	11	984
	WFMP	80	1	3
Spring	AMQP	408	33	108
	ANDROID	305	2	11
	BATCH	1,732	33	432
	BATCHADM	243	4	20
	DATACMNS	604	33	158
	DATAGRAPH	848	4	60
	DATAJPA	330	38	147
	DATAMONGO	622	40	271
	DATAREDIS	551	17	49
	DATAREST	414	23	132
	LDAP	566	5	53
	MOBILE	64	3	11
	ROO	1,109	15	714
	SEC	1,618	42	541
	SECOAUTH	726	7	101
	SGF	695	19	107
	SHDP	1,102	9	45
	SHL	151	3	11
	SOCIAL	212	4	15
	SOCIALFB	253	5	15
	SOCIALLI	180	1	4
	SOCIALTW	153	5	8
	SPR	6,512	12	130
	SWF	808	20	134
	SWS	925	25	174
Total		61,431	690	9,459

3.2. Study Design

Our preliminary study focuses on addressing the following research question:

- **RQ1**: Do different operator combinations have different impacts on the performance of bug localization?

To answer the RQ1, we apply every possible combina-

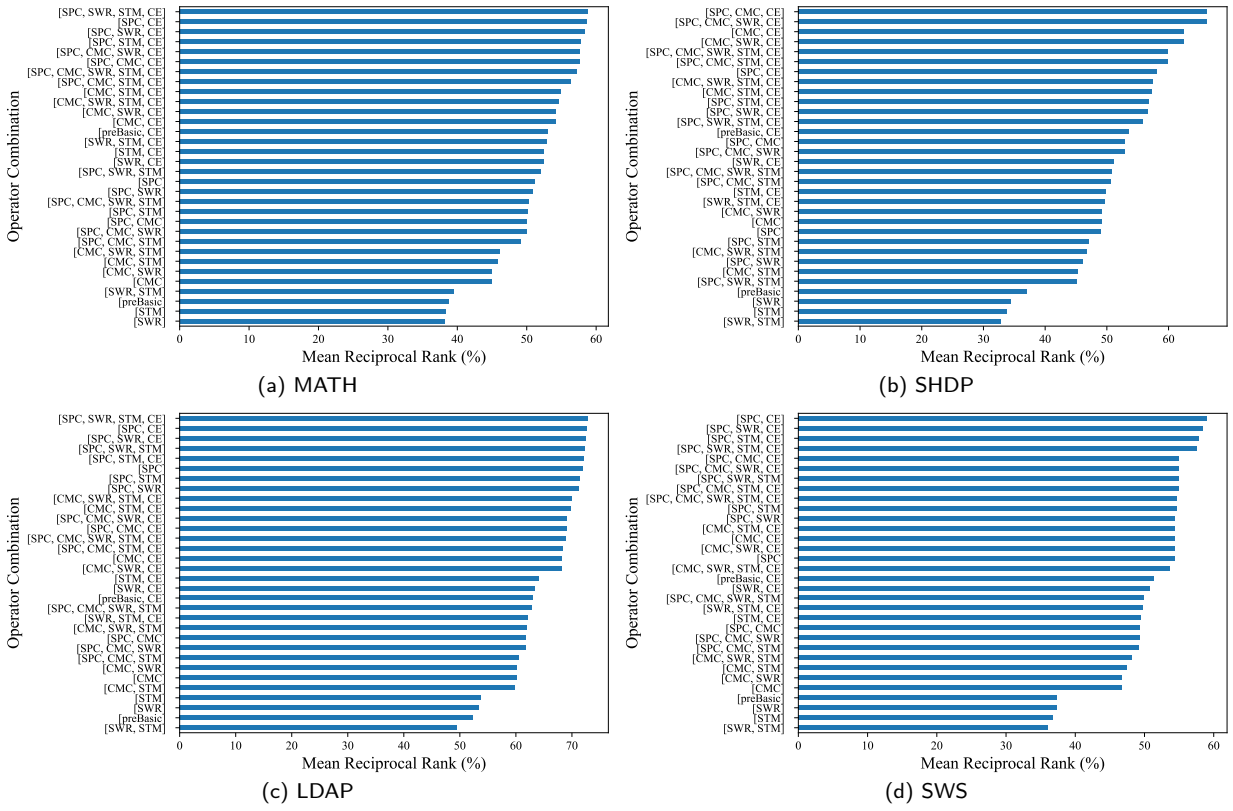


Figure 2: Results (MRR) of four sampled subjects for every combination.

tion of pre-processing operators as shown in Table 1 on a specific technique named BugLocator [26] to check if they actually affect the bug localization approach. Among them, `preBasic` is always turned on for any subject since the tokenization step is necessary for building input queries. We then apply all the combinations of pre/post-processing operators (5 pre-processing and 1 post-processing) to every subject. We prepare 32 ($=2^5$) different combinations by alternatively activating and deactivating each different operator with the other five operators. Each pre-processing operator is applied to all the bug reports and source code files before feeding them to bug localization techniques, while the post-processing operator (if it is activated in the combination) is applied after obtaining a ranked list of suspicious files.

To implement a bug localization pipeline, DIGBUG is built based on the Vector Space Model (VSM) [31] to conduct IRBL. *The choice of using this model is based on the performance results [20] that classified VSM as the best among all the IR-classification techniques including Latent Semantic Analysis (LSI) and Latent Dirichlet Allocation (LDA).* VSM computes the similarity between tokens of an incoming bug report and tokens of each source code file. While existing techniques apply more advanced modelization techniques (e.g., rVSM in BugLocator [26]), we use VSM as a baseline technique in our study to focus on unequivocally highlighting the impact of pre/post-processing operator combinations.

3.3. Results

One of the primary findings of our preliminary study is that each combination of operators[2] shows different performances. To check the performance difference on the processing operators, we applied different operators to an existing approach named BugLocator [26]. We applied all the pre-processing operator combinations, and it already showed a performance range from 38.2% to 39.9% for MAP and from 50.6% to 52.5% for MRR, respectively. This results indicate that operator combinations has impact on existing techniques even only with pre-processing combinations for overall results. Further experiment driven by the former founding includes all the 32 ($=2^5$) operator combinations and we apply them to sample subjects. For example, the localization performance for the subject MATH, ranged from 38% to 59% whereas the best one is achieved with `SPC`, `SWR`, `STM`, and `CE` operators while the worst one is with `SWR` as shown in Figure 2(a). To be explicit, the best operator combination of the sample projects we have; MATH, SHDP, LDAP, SWS are [`SPC`, `SWR`, `STM`, `CE`], [`SPC`, `CMC`, `CE`], [`SPC`, `SWR`, `STM`, `CE`], and [`SPC`, `CE`] respectively. The results from other subjects are similar to the one for MATH. In particular, the MRR value of the best combination for SHDP has improved 33.3 percentage points (32.9% → 66.2%) against the worst one.

Similarly, in our baseline localization technique, it achieves the best performance for MATH when all the operators ex-

[2] In our results, the `preBasic` indicates the results from VSM without any pre/post-processing involved in.

Table 3
MAP and MRR for sample subjects (single version matching [18] for projects).

Subject	BugLocator [26]		BRTracer [7]		BLUiR [6]		AmaLgam [29]		BLIA [28]		Locus [27]		Pre-study	
	MAP	MRR	MAP	MRR	MAP	MRR	MAP	MRR	MAP	MRR	MAP	MRR	MAP	MRR
MATH	0.1563	0.2173	0.1586	0.2274	0.1952	0.2413	0.2122	0.2627	0.1765	0.2394	0.1895	0.2251	**0.4376**	**0.5882**
SHDP	0.4433	0.6279	0.4652	**0.6734**	0.3899	0.5184	0.3897	0.5184	0.4654	0.6222	0.4633	0.5826	**0.4835**	0.6617
LDAP	0.4401	0.6344	0.4875	0.7197	0.4681	0.6251	0.4681	0.6251	0.4824	0.6665	0.3857	0.5058	**0.5238**	**0.7286**
SWS	0.4002	0.5400	0.4211	0.5872	0.3811	0.4886	0.3811	0.4886	0.3969	0.5456	0.4177	0.5680	**0.4317**	**0.5898**
Average	0.3600	0.5049	0.3831	0.5519	0.3586	0.4684	0.3628	0.4737	0.3803	0.5184	0.3641	0.4704	**0.4692**	**0.6421**

cept CMC are activated. However, for SWS, the best performance is retrieved when SPC along with CE is activated. The operator combination of SPC, SWR, and CE performs better for MATH, LDAP, and SWS. In contrast, in the case of SHDP, such combination performs 9.5% less in terms of MRR. This confirms our hypothesis that the "one-combination-fits-all" strategy is not the best option for performance.

Table 3 shows the results of our preliminary study comparing with other IRBL tools for the four sampled subjects. Except for the MRR of SHDP, our approach outperforms all the existing techniques. Overall, this preliminary study reported an average improvement of up to 11.0% and 17.4% for MAP and MRR, respectively.[3]

> *Our preliminary study results indicate that considering different operator combinations significantly affects the bug localization performance on the subjects. This answers our RQ1.*

Based on the results of this study, we motivate the following hypothesis: **different combinations of operators may lead to a better bug localization if we can identify the relationships between the characteristics of subjects and operators**. Here, the challenge is how to find the best performing operator combination for each subject. In the above preliminary study, the unit of the subject is coarse-grained (i.e., project), but it is also able to search for the best combination for a finer-grained subject (i.e., bug report). This forms the basis of our motivation to study the possible operator combination that performs best for a subject. To this end, we describe our tool DIGBUG that selects the best combination of operators for different bug reports in Section 4.

4. DIGBUG — Operator Selector for IRBL

This section describes DIGBUG, a tool for selecting a combination of pre/post-processing operators used in IRBL. Based on the motivation discussed in Section 3, this tool first identifies the characteristics of the incoming bug report. These characteristics allow the tool to apply the correct combination of operators for a bug report. Finally, the tool compares the similarity between the bug reports and source code files and provides a ranked list of suspicious code files.

DIGBUG consists of two phases as shown in Figure 3: (1) Training[4] and (2) Localization. The intuition here is that it is possible to figure out the best performing combination of pre/post-processing operators if we can characterize the bug reports. Besides, we make two assumptions (1) the bug reports can be characterized by their attributes such as the presence of stack trace, attachment, and type of reporter, etc. (2) the bug reports can be grouped by computing the similarity between them based on the attributes.

Table 4
Features used for characterizing bug reports.

Code Entity	Is any code entity available in the report?
Description	Is a description provided?
Stack Trace	Does the report contain any stack trace?
Developer or User	Whether the report is submitted by a developer or user?
Attachment	Does the report have an attachment?

In the *training* phase, the tool takes a set of bug reports as training data, and extracts attributes from each report (Section 4.1). The attributes characterize a bug report, and DIGBUG decides a bucket (B_k) where a bug report belongs, based on its attributes. The tool searches for the best combination of pre/post-processing operators (OC_k) for each bucket by applying every combination exhaustively.

The *localization* phase starts with an incoming bug report, which is the input of bug localization. In this phase, DIGBUG first extracts attributes of the report used to determine the bucket where the report belongs. Since the training phase already identified the best combination, the tool applies the identified operator combination to the bug localization of the report. The operators are used before (pre-processing) and after (post-processing) applying any IRBL technique to subjects (bug reports). In this work, we use a baseline bug localization with VSM as described in Section 3.2. As a result, DIGBUG gives a ranked list of suspicious source code files corresponding to the incoming report.

4.1. Attribute Extraction

To characterize bug reports, DIGBUG uses five attributes as shown in Table 4. These are all binary attributes and make 32 combinations, which will correspond to each bucket later. We select these attributes since they are already adopted by existing techniques [27, 7, 28] to characterize bug reports and improve the bug localization performance.

The extraction of the attributes follows the below heuristics. *Code Entity* is positive if a bug report has any camel-case word or parenthesis; and we have *Code Entity* database to check since the source code should be already exists. If the

[3] This is the results for preliminary study.

[4] This training process can be regarded as '*searching*' rather than that of machine learning since it tries every possible combination of operators. Nevertheless, we use the term '*training*' since it depends on the training data.

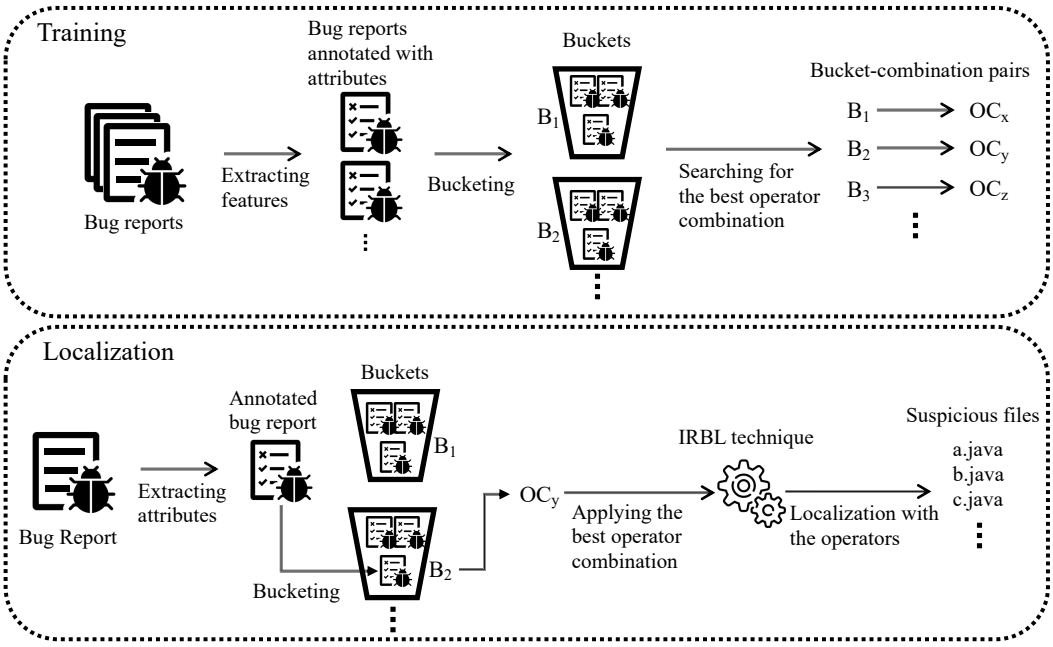


Figure 3: Overall procedure for training and localization of DIGBUG.

description section of a bug report is not empty, the *Description* attribute is on for the bug report. *Stack trace* is positive if any stack trace is given for a bug report. To accurately extract the stack traces from the bug report, we utilized the tool *infoZilla* [44]. *infoZilla* is a library that helps extract structural data from unstructured data sources such as bug reports. This tool is a well-known method to extract Stack Traces with an accuracy of 98.5%.

To identify whether the *submitter* of a report is a developer or user, the tool looks up the revision history of the subject where the report belongs. If the submitter has committed any changes to the subject, our tool regards the submitter as a developer; otherwise, it is a user. The *attachment* information can be directly accessed by looking up the metadata of a bug report.

4.2. Bucketing

Based on the attributes of each bug report, DIGBUG classifies bug reports into 32 buckets as described in Section 4.1 (i.e., five binary attributes and their combinations). As shown in Figure 3, each bucket ($B_1, B_2, \ldots$) contains bug reports with the same characteristics based on the five attributes. We assume that an identical operator combination can treat bug reports in a bucket.

4.3. Searching for the best-performing combination

Once bug reports in training data are assigned to different buckets, DIGBUG identifies the best operator combination for each bucket by trying to apply all possible combinations to the reports in a bucket exhaustively. We use MRR as the metric to compare different combinations and figure out the best one.

Algorithm 1 shows in detail the training phase of DIGBUG. As a function, the phase takes five input arguments and produces a map (BC) that tells us which operator combination performs best for a given bucket. The input arguments are a set of bug reports (R) for a training purpose, a set of available pre/post-processing operators (O), a function f that extracts attributes from a bug report, and a bug localizer to be used for evaluating the performance of an operator combination ($oc \in 2^O$) for a given set of reports in a bucket. As shown in the algorithm, DIGBUG first extracts attributes of each bug report in R (Line 3) and then assigns them (Line 5) into different buckets ($b \in B$ and $|B| = 2^k$ where k is the number of attributes). After obtaining a set of operator combinations (Line 7), the tool exhaustively searches for the best operator combination for every bucket (Lines 10–16) and returns the result (Line 17).

4.4. Localization

Once the pair of buckets and operator combinations are trained, DIGBUG takes a newly submitted bug report and provides the best combination of pre/post-processing operators for the report in the localization phase. Furthermore, in this phase, the tool first extracts attributes from the incoming bug report to find its corresponding bucket and then retrieves the best combination found in the training phase. The retrieved combination is plugged into an IRBL technique. The operators in the combination are applied in the pre/post-processing steps of the technique resulting in the list of suspicious files.

Algorithm 1: Training phase of DIGBUG.

Input: a set of bug reports (for training): $R = \{r_1, r_2, \ldots, r_n\}$
Input: a set of operators: $O = \{o_1, o_2, \ldots, o_m\}$
Input: attribute extraction function: $f : R \rightarrow \{0,1\}^k$
Input: bug localizer: $localizer : R \times 2^O \rightarrow P(S)$ where $P(x)$ is permutations of a set x and S is a set of source code files in a subject.
Input: bucketing function: $bk : R \times \{0,1\}^k \rightarrow B$
Output: a map of a bucket and an operator combination: $BC = \{(b_i, OC_x), (b_j, OC_y), \ldots, (b_m, OC_z)\}$

```
Function train(R, f)
    // produce a map of report (r_i ∈ R) and attribute tuple (f_i ∈ {0,1}^k).
    R_f := R.map{(r_i, f(r_i))} ;
    // produce a map of report (r_i ∈ R) and bucket (b_j ∈ B).
    R_b := R.map{(r_i → bk(R_f(r))};
    // obtain all combinations of operators.
    OC = C(O);
    // output: bucket – operator combination map.
    BC = ∅;
    foreach ∀b_i ∈ B do
        // collect all bug reports belonging to b_i.
        R_i := R.filter(R_b(r) == b_i);
        // find the best combination ('MRR' is a function computing MRR values defined in Section 2.3).
        OC_best := argmax_{oc_j ∈ OC} {MRR_{∀r ∈ R_i} {localizer(r, oc_j)}};
        // add the best combination of the given bucket.
        BC := BC + (b_i → OC_best);
    return BC;
```

5. Evaluation

This section describes the experimental setup, dataset for training, and localization used to assess the performance of DIGBUG and report its results. The aim is to investigate if the selection of a particular operator combination impacts bug localization performance when considering different bug reports.

5.1. Experimental Setup

The outcome of our preliminary study and its results allow us to propose further research questions:

- **RQ2**: Is the best operator combination different for an individual bucket decided by the characteristics?
- **RQ3**: Does our bucketing approach provide significant improvement over the state-of-the-art?

In this evaluation, to implement the approach described in Section 4, we use the following configuration. First, the five operators (four for pre and one for post processing) listed in Table 1 to generate the set of operator combinations (note that `preBasic` is always activated as a baseline operator, and thus it is not included in the generation of combinations). Furthermore, the same configuration used for the bug report is applied to the source code within the dataset. Second, we leverage the same IR-technique (i.e., VSM) as with the preliminary study described in Section 3 since our objective is to focus on investigating the impact of operator selection on bug localization. In practice, any IRBL technique can be plugged into DIGBUG instead of the baseline localizer.

Table 5
Number of features used for characterizing bug reports.

Code Entity	5,191
Description	9,459
Stack Trace	1,137
Reporter as Developer	2,511
Reporter as User	6,948
Attachment	6,210

We address our RQ3 by considering the results of the six popular IRBL techniques [26, 7, 6, 29, 28, 27], which are reported in Bench4BL [18]. The results are obtained with the following configuration (1) single version matching and (2) test files included, which are listed in the literature Bench4BL [18]. We take the same condition to make the comparison fully fair. The former is selected for simplicity since multi-version matching increases the complexity of the evaluation. The latter is chosen as Bench4BL's results suggested [18].

5.2. Dataset

To evaluate our approach, we leverage two different datasets: one from D&C [8] that has approximately 20K bug reports for the training and the other from Bench4BL [18] for localization (i.e., evaluation). The training phase was conducted using different dataset due to the lack of the number of bug reports for the test, and we devised as this cross-project setting generalizes the approach. We leverage this step to segregate/separate the buckets based on the attributes. This training allows generalizing the attributes for each bucket.

Training dataset: D&C dataset has 19,600 bug reports and furthermore contains additional subjects (e.g., Apache WICKET [45]). This dataset is curated by considering only bug reports since the Bench4BL dataset contains all the issue reports, including bug reports and discarding every report that is not completely paired with source code. The bug reports that are considered in the dataset are the ones marked as `RESOLVED`, `FIXED` or `CLOSED`. Because of the curation phase, D&C ends up having 4,467 intersections (i.e., the same bug reports as from Bench4BL) of the bug reports. We discard them from the training phase to generalize the buckets and their corresponding operator combinations. Finally, we ended up having 15,133 bug reports in the training phase for our approach.

Dataset for localization To assess the approach, we use the dataset that consists of bug reports and source code pairs from Bench4BL [18]. The use of source code and bug report pairs allows training for the retrieval of potential buggy files. The dataset has 9,459 bug reports in total and 61,431 source code to index. We use all the 9,459 bug reports to predict the potential buggy source files in the localization phase. Table 5 shows the numbers for each feature extracted from the bug reports. We hypothesized that there are bug reports that do not contain the body (i.e., description) since we found several ones in practice (e.g., [46]). However, all of the Bench4BL's bug reports include descriptions. While recent IRBL studies often use outdated datasets (so-called

Table 6
Bucket – Operator combinations and the number of reports per each bucket found in this study with the setup described in Sections 5.1 and 5.2.

Index	Bucket by Attributes	Operator Combination	# of Reports
1	code-entity, developer, attachment, description, stack-trace	SPC, CE	74
2	code-entity, developer, attachment, description	SPC, CE	740
3	code-entity, developer, attachment	N/A	
4	code-entity, developer, description, stack-trace	SPC, CE	79
5	code-entity, developer, description	SPC, SWR, STM, CE	402
6	code-entity, developer, stack-trace	N/A	
7	code-entity, developer	N/A	
8	code-entity, user, attachment, description, stack-trace	SPC, CE	332
9	code-entity, user, description, stack-trace	SPC, SWR, STM, CE	194
10	code-entity, user, attachment, description	SPC, SWR, STM, CE	2280
11	code-entity, user, attachment	N/A	
12	code-entity, user, attachment, stack-trace	N/A	
13	code-entity, user, stack-trace	N/A	
14	code-entity, user, description	SPC, STM, CE	1090
15	code-entity, user	N/A	
16	developer, attachment, description, stack-trace	SPC, CMC, SWR	57
17	developer, attachment, description	SPC, CMC, SWR	666
18	developer, description, stack-trace	SPC	51
19	developer, description	SPC, CMC	442
20	developer, stack-trace	N/A	
21	developer, attachment	N/A	
22	developer, attachment, stack-trace	N/A	
23	user, attachment, description, stack-trace	SPC, SWR, STM	222
24	user, description, stack-trace	SPC	129
25	user, attachment, description	SPC, CMC, SWR	1839
26	user, description	SPC, CMC, STM	869
27	user, attachment	N/A	
28	user	N/A	
29	developer	N/A	
30	code-entity	N/A	
31	attachment	N/A	
32	stack-trace	N/A	

"old subjects" [18]) to evaluate their techniques, we employ Bench4BL since the corresponding study based on the benchmark shows that the latest subjects are more effective and reliable to assess IRBL techniques.

5.3. Experimental Results

As a result of the training phase of our experiment, we obtain a list of buckets that consist of the attributes of the bug reports. Furthermore, it also results in the best operator combination for individual buckets. A list of the attributes, its operator combination, and the number of bug reports for each bucket are presented in Table 6. In our result, there are cases where the operator combinations are not available due to the lack of corresponding attributes of the bug reports[5].

We found that applying incorrect operator combinations for a particular bucket leads to inaccurate localization. The results imply that applying an operator `CMC` (i.e., CamelCase splitting) on "code entity" would potentially change the entity itself, leading towards lower accuracy for the localization. A stack-trace is a list of the method calls captured in the middle of the execution of the application. This stack-trace is neither a natural language free-form text nor the standard source code. We discovered that bug reports with only stack-trace included should be treated with only the `SPC` operator. This means other operators (i.e., `SWR`, `CMC`, `STM`) impediment for the accurate localization since stack-trace likely consist of information relevant for debugging, applying the operators would alter its meaning and context. For example, stack-trace may have the keyword "..connections.." which, upon application of `STM` would change to "connect" thereby changing the context of the stack-trace. The operator `SPC` is constantly being applied, and it indicates that special characters (e.g., #, $ and %) need to be removed as they do not provide meaningful information for the localization. Another attribute, such as the CE (code-entity), is applied to every bucket with any form of code-entity present to extract practical terms from the code, such as the class names, method names, and parameter types. As a direct result, each bug report requires a different operator combination based on its attributes for accurate IRBL performance. This supports our main claim that a one-size-fits-all does not fit every bug report.

> *RQ2 is addressed by the results that demonstrate bug reports can be classified into different buckets by their attributes, and the best operator combination is different for an individual bucket.*

Table 7 shows the aggregated MAP and MRR for all the subjects described in Table 2. It shows that DIGBUG achieves better results than the existing state-of-the-art techniques. The improvements are up to 8.0% and 12.6% of MAP and MRR, respectively for the aggregated results. It also indicates that DIGBUG can achieve at least 3.5% and 4.4% improvement against the best-known results from Locus and BRTracer. Additionally, we use the Mann-Whitney U test [47] to identify whether the differences are significant. This statistical test is applied between DIGBUG and each technique. If the result has a p-value lower than 0.01, it is indicated by a single asterisk. If it is lower than 0.001, then it is denoted with a double-asterisk. The test results clearly show that the differences are mostly significant except for those of the Locus [27].

Table 8 shows the results of our approach for all the evaluated subjects. The results in the table are project-independent, unlike Table 7. In the majority of the cases, our approach outperforms the six state-of-the-art techniques. On average, DIGBUG provides improvements from 6.8% to 9.6% and 9.4% to 14.9% for MAP and MRR, respectively. These results show that application of different operator combinations brings better localization results based on different buckets of bug reports. Additionally, it indicates that the cross-project training and localization setting works where insufficient data is available for the training.

[5]For example, Index 32 of Table 6 as it cannot be a bug report with just stack-trace inside it; Index 6 of Table 6 as there is no bug report with those particular set of attributes within our dataset.

Table 7
Summary of MAP/MRR of IRBL techniques for the subjects (aggregated results).

	BugLocator	BRTracer	BLUiR	AmaLgam	BLIA	Locus	DigBug
MAP	0.3052*	0.3330*	0.2881**	0.2906**	0.3014*	0.3289	**0.3681**
MRR	0.4223**	0.4690**	0.3869**	0.3899**	0.4155**	0.4430	**0.5134**

*: p-value < 0.01, **: p-value < 0.001

Table 8
MAP and MRR for each subject listed in Table 2 (single version matching for projects).

Subject	BugLocator		BRTracer		BLUiR		AmaLgam		BLIA		Locus		DigBug	
	MAP	MRR	MAP	MRR	MAP	MRR	MAP	MRR	MAP	MRR	MAP	MRR	MAP	MRR
CAMEL	0.3235	0.4621	0.3646	0.5270	0.3005	0.4188	0.3032	0.4210	0.3097	0.4451	**0.3986**	**0.5571**	0.3355	0.4816
HBASE	0.2993	0.4168	0.3463	0.4884	0.2818	0.3938	0.2820	0.3942	0.3094	0.4258	0.3084	0.4059	**0.3678**	**0.5198**
HIVE	0.2693	0.3670	0.3178	0.4521	0.2769	0.3914	0.2772	0.3916	0.2412	0.3312	**0.3310**	**0.4580**	0.2670	0.3921
CODEC	0.6227	0.8341	0.6333	0.8199	**0.7011**	0.8625	0.7011	0.8625	0.6914	0.8829	0.3519	0.4087	0.6574	**0.8953**
COLLECTIONS	0.2318	0.3327	0.2319	0.3457	0.2174	0.2659	0.2191	0.2659	0.2493	0.3606	0.2670	0.3090	**0.5567**	**0.8532**
COMPRESS	0.5637	0.7545	0.5747	**0.7976**	0.4874	0.6382	0.4816	0.6345	0.5797	0.7822	**0.5872**	0.7801	0.5377	0.7452
CONFIGURATION	0.0378	0.0480	0.0385	0.0530	0.0413	0.0502	0.0413	0.0502	0.0345	0.0476	0.0299	0.0430	**0.6082**	**0.8143**
CRYPTO	0.1622	0.2665	0.1711	0.3044	**0.1982**	**0.3974**	0.1982	0.3974	0.1588	0.3616	0.1118	0.3421	0.1686	0.3105
IO	0.7640	0.8574	0.7508	**0.8809**	0.6797	0.7332	0.6784	0.7309	**0.7744**	0.8489	0.4053	0.4322	0.5689	0.6826
LANG	0.5371	0.6446	0.5426	0.6441	0.5327	0.5783	0.5367	0.5810	0.5349	0.6423	0.5810	0.6181	**0.7556**	**0.8936**
MATH	0.1563	0.2173	0.1586	0.2274	0.1952	0.2413	0.2122	0.2627	0.1765	0.2394	0.1895	0.2251	**0.4422**	**0.5971**
WEAVER	0.6212	0.6666	0.6331	0.7500	0.6637	0.6666	0.6637	0.6666	0.5695	0.6666	0.4996	0.5500	**0.7679**	**1.0000**
CSV	0.6104	0.6677	0.6108	0.6784	0.6075	0.6369	0.6313	0.6845	0.4048	0.6429	0.6554	0.6970	**0.7872**	**0.9167**
ENTESB	0.0559	0.0511	0.0542	0.0624	**0.0652**	**0.0775**	0.0652	0.0775	0.0314	0.0638	0.0555	0.0704	0.0491	0.0682
JBMETA	0.2442	0.4387	0.2351	0.4312	0.1805	0.3233	0.1832	0.3258	0.2078	0.3698	0.2534	0.3639	**0.2913**	**0.4601**
ELY	0.0587	0.1506	0.0668	0.1587	0.1098	0.1962	0.1098	0.1962	0.0939	0.1667	**0.1260**	0.1667	0.1161	**0.2118**
WFARQ	0.5000	0.5000	0.3333	0.3333	1.0000	1.0000	1.0000	1.0000	0.1111	0.1111	0.5000	0.5000	**1.0000**	**1.0000**
WFCORE	0.3202	0.4444	0.3326	**0.4687**	0.2552	0.3392	0.2565	0.3402	0.2830	0.3834	**0.3607**	0.4607	0.3389	0.4414
WFLY	0.2283	0.3196	0.2572	0.3623	0.2099	0.2972	0.2104	0.2982	0.2118	0.2994	0.2562	0.3500	**0.2628**	**0.3736**
WFMP	0.4534	0.5833	0.2675	0.2398	0.6818	0.8333	0.6818	0.8333	0.5000	0.6667	0.6226	0.7778	**0.8254**	**1.0000**
SWARM	0.2936	0.3852	0.3326	0.4317	0.3412	0.3960	0.3412	0.3960	0.2708	0.3608	0.2651	0.3656	**0.3664**	**0.4638**
AMQP	0.4426	0.6205	**0.4775**	**0.6754**	0.4196	0.5691	0.4210	0.5694	0.4626	0.6661	0.4533	0.6229	0.4245	0.6447
ANDROID	0.3626	0.5121	0.3536	0.5904	**0.3761**	0.4747	0.3761	0.4747	0.3138	0.5076	0.0796	0.0698	0.3660	**0.6894**
BATCH	0.3186	0.4848	0.3284	0.5023	0.2805	0.3941	0.2933	0.4124	0.3158	0.4735	0.3664	0.5449	**0.3721**	**0.5662**
BATCHADM	0.3218	0.4325	0.3479	0.5011	0.3657	0.4806	0.3657	0.4806	0.4313	0.5724	0.4322	**0.6252**	**0.4458**	0.5869
DATACMNS	0.4565	0.5916	0.4683	0.6429	0.4581	0.5632	0.4581	0.5632	0.5133	0.6670	**0.5244**	0.6482	0.5134	**0.6701**
DATAGRAPH	0.1519	0.2384	0.1592	0.2497	0.1714	**0.2788**	**0.1718**	0.2788	0.1426	0.2577	0.1609	0.2578	0.0116	0.0690
DATAJPA	0.4822	0.6649	0.4892	0.6854	0.4769	0.6170	0.4767	0.6170	**0.5311**	**0.7098**	0.4769	0.6329	0.4975	0.6836
DATAMONGO	0.4582	0.6344	0.5095	**0.6965**	0.4519	0.5769	0.4519	0.5769	**0.5212**	0.6807	0.4861	0.6322	0.3783	0.5231
DATAREDIS	0.5353	0.7627	0.5561	0.8081	0.5801	0.7626	**0.5813**	0.7637	0.5717	**0.8082**	0.4999	0.7259	0.4493	0.6761
DATAREST	0.3492	0.5458	0.3939	**0.6372**	0.3550	0.5125	0.3584	0.5162	0.3651	0.5730	0.3802	0.5974	**0.3956**	0.5722
LDAP	0.4401	0.6344	0.4875	0.7197	0.4681	0.6251	0.4681	0.6251	0.4824	0.6665	0.3857	0.5058	**0.5291**	**0.7335**
MOBILE	0.6909	0.8864	0.7116	0.9545	**0.9224**	**1.0000**	0.9224	1.0000	0.7285	0.8939	0.5042	0.5862	0.7855	0.9545
ROO	0.1164	0.1628	0.1293	0.1821	0.0910	0.1283	0.0928	0.1297	0.1001	0.1422	0.1208	0.1811	**0.3360**	**0.4353**
SEC	0.3209	0.4237	0.3368	0.4438	0.3087	0.3736	0.3133	0.3788	0.3502	0.4496	0.3145	0.3857	**0.4922**	**0.6300**
SECOAUTH	0.1983	0.3659	0.2128	0.3965	0.1381	0.2620	0.1430	0.2714	0.1891	0.3522	0.1990	0.3683	**0.3205**	**0.5542**
SGF	0.4173	0.6546	0.4223	0.6850	0.3591	0.5973	0.3589	0.5970	0.3682	0.6174	**0.4359**	**0.7245**	0.4039	0.7026
SHDP	0.4433	0.6279	0.4652	**0.6734**	0.3899	0.5184	0.3897	0.5184	0.4654	0.6222	0.4633	0.5826	**0.4745**	0.6631
SHL	0.2533	0.4037	0.2621	0.4166	0.2827	0.4220	0.2828	0.4221	0.2836	0.4015	**0.3251**	0.4579	0.2992	**0.5121**
SOCIAL	0.6110	0.6937	0.5900	0.6726	0.1979	0.2245	0.2496	0.3000	0.5285	0.5689	**0.6569**	**0.7029**	0.5888	0.6541
SOCIALFB	0.5541	0.6416	**0.6156**	**0.7401**	0.4818	0.6167	0.4818	0.6167	0.4064	0.5301	0.5382	0.6929	0.4849	0.6234
SOCIALLI	0.4711	0.6250	**0.6384**	0.7083	0.3929	0.3958	0.4504	0.4166	0.2989	0.3208	0.4081	0.6875	0.5837	**0.8750**
SOCIALTW	0.7382	0.7937	0.6750	0.7292	0.3814	0.4271	0.5014	0.5833	0.5594	0.6188	0.5456	0.6250	**0.9009**	**1.0000**
SPR	0.3074	0.4684	**0.3377**	**0.5165**	0.2061	0.3284	0.2182	0.3386	0.2878	0.4319	0.0169	0.0241	0.3021	0.4721
SWF	0.3812	0.4758	0.3974	0.5060	0.3647	0.4579	0.3613	0.4548	0.4038	0.5015	**0.4384**	**0.5502**	0.3174	0.4548
SWS	0.4002	0.5400	0.4211	0.5872	0.3811	0.4886	0.3811	0.4886	0.3969	0.5456	0.4177	0.5680	**0.4460**	**0.6093**
Average	0.3821	0.5064	0.3922	0.5299	0.3767	0.4746	0.3836	0.4827	0.3644	0.4930	0.3649	0.4757	**0.4603**	**0.6234**

The highlighted values in purple (n.nnnn) and green (n.nnnn) background denote the highest MAP and MRR, respectively for each project.

For example, DIGBUG's result for COLLECTIONS subject achieves up to 34% and 58.7% points against the worst ones, while the result for the SOCIALTW subject shows improvements of 30% and 40% points in terms of MAP and MRR, respectively. Note that other existing techniques often use a single set of pre/post-processing operators without considering the different characteristics of each bug report. Thus, some necessary tokens can be remove, or other unnecessary tokens may not be filtered out during pre/post-processing. Our bucketing approach can assign an appropriate set of the operators, and this may lead to better performance. We can expect performance improvement with a more precise bucketing technique.

The result for the subject DATAGRAPH shows that DIGBUG achieves the worst results as compared to others. We manually investigate the reason for this and found that 12 out of the 60 bug reports contain code entities in them. A more detailed check of the bug report reveals that the keyword "QueryResult" often appears in them. As the subject is related to the popular NoSQL language, it is natural for the users to mention this keyword, thereby populating the bug report with the keyword that generates noise for our dataset. The key takeaway is to be careful when considering bug reports that may contain such 'noisy' keywords.

It is well known that the presence of code entities (e.g., method or class names) within a bug report improves the performance of the localization [27]. However, our results from DIGBUG provide new insights; for subjects such as ENTESB, SHL, SOCIALTW, WEAVER, applying the CE operator does not significantly impact the performance. Furthermore, the best performance for specific subjects such as BATCHADM, CSV, and DATAGRAPH are retrieved when the post-processing operator CE is deactivated.

Additionally, we observed that what issue reports are successfully localized. Among a total of 9,459 reports, 5,184 (i.e., 54.80%) has code entities, and most of them are written by developers to prioritize after pre-processing steps and analysis. 82.99% (i.e., 4,302 reports) of the corresponding 5,184 reports are successfully ranked in at least the top 10. This implies that code entities being provided by developers are helpful for better performance on bug localization as stated in [27].

> *Addressing RQ3, the evaluation results indicate that applying different operator combinations to different buckets improves the performance of bug localization over the state-of-the-art techniques.*

5.4. Threats to validity

As in any empirical assessment, our study bears some threats to validity.

External validity: Our experiments examine only Java subjects. However, the same process in the study can be applied to other subjects that are implemented in other programming languages. Another threat to the validity of our study is that our subjects are all based on an open-source development model. The practice in the software industry may involve projects with specific characteristics that may affect the performance of the IRBL techniques with varying levels of success. The evaluation of the approach has been conducted with "Single version" of each subject. Bench4BL [18] already shows that considering multiple versions of subjects can derive better performance. However, our goal is to show the varying impact of the operator combinations (i.e., processing strategy). Therefore, considering multi-version is outbound of the approach. Although we applied the combinations of the operators to the baseline (i.e., VSM) technique and it outperforms the other techniques, it may retrieve different impact if we apply such operators on each technique. It may depend on an experimental environment or specific characteristics.

Internal validity: We use our implementations for the different pre-processing operators, which may carry some limitations. We also relied on the simple VSM baseline for the comparison to determine the effect of pre/post-processing operators. The latter, however, is commonly used in the literature, which mitigates the threat to validity. It should be noted that the training process of DIGBUG could be time-consuming and might require significant resources, but it is a one-time cost. Although there would be changes towards the software maintenance over time, the need for re-training would be relatively low as the attributes required for DIGBUG would be retained.

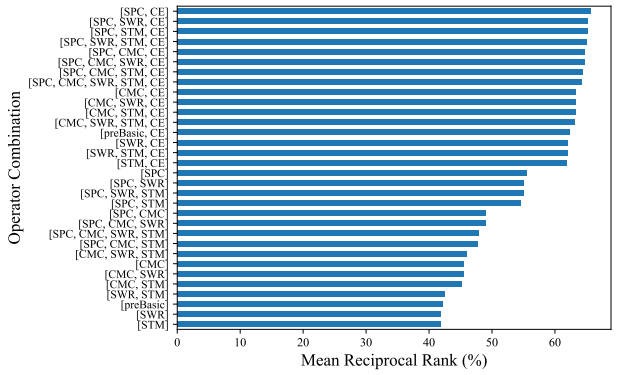
If the number of buckets and possible combinations increase, the computation cost would be much higher. It would be easy to make the process parallel and the potential improvement in the localization phase can compensate for the cost.

5.5. Discussion & Future work

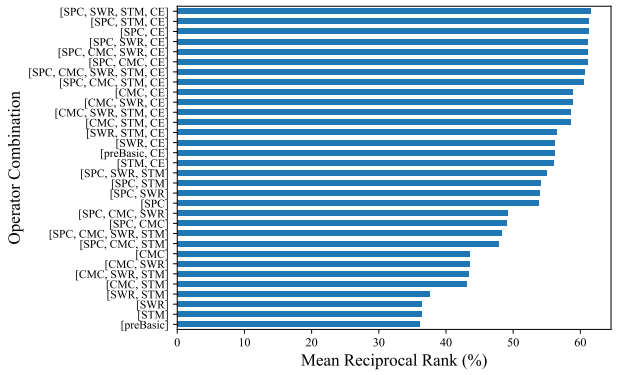
Pre/post-processing operators: In this study, we use data processing operators that are commonly used in state-of-the-art bug localization techniques and well-known for text retrieval. Figure 4 shows the MRR results for the sampled buckets (Index# 2, 9, and 25 in Table 6) from our experiment (shown in Tables 7 and 8 of Section 5). These buckets classify the bug reports based on their attributes without considering the subjects. When applying the different operator combinations to each bucket, the results vary for each combination. As the results indicate, if an unsuitable operator combination processes a bug report, it significantly decreases the performance.

Categorization of bug reports: For the categorization of the bug reports, we focused on the attributes of the available reports in most issue tracking systems. Our future work includes discovering additional attributes from bug reports that would potentially lead to a better categorization.

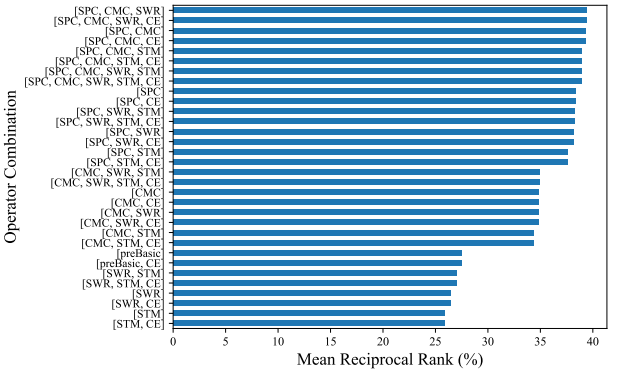
Information retrieval model: We use VSM (i.e., the most basic one for IR-based bug localization) instead of other complex models such as deep learning techniques. Although the deep learning techniques may show better results, our goal is to show that different operator combinations should be applied depending on the characteristics of bug reports and even subjects.



(a) Bucket Index 2



(b) Bucket Index 9



(c) Bucket Index 25

Figure 4: Results (MRR) of three sampled buckets for every combination.

Furthermore, deep learning based approaches still lag compared to the time of the classical approaches. For instance, Lam et al. [11] presented a simple deep learning approach that combines rVSM and DNNs (Deep Neural Networks) where they reported the time of a couple of minutes to retrieve one prediction (i.e., results for a bug). When compared to DIGBUG, it takes around 0.6 seconds on average for retrieving one prediction. Additionally, the performance of the former provides a limited improvement of 2.8% points in terms of MAP against that of the LR [13], which is a VSM-based approach that integrates functional decompositions, bug-fixing history, API descriptions, and code change history for bug localization.

However, it will be worth comparing with approaches such as DeepLoc [48] or DreamLoc [43] since such approach also shows a significant improvement compared to an approach (i.e., BugLocator [26] and BRTracer [7]) in our experiment, even though it used a different dataset than ours. It will guide if our approach also has a significant impact on learning-based approaches.

Classifying bug reports: Our study assumes that the bug reports in the benchmark [18] are correctly classified as "bug". However, DIGBUG might not be effective if the classification is incorrect. Antoniol et al. [49] discovered that many issue reports are labeled as "bug" even though they are enhancements, refactoring/restructuring, and organizational issues due to lack of better classification support. For example, a simple organizational issue may contain stack traces in a report. Thus, the classification accuracy may affect the performance of DIGBUG.

Applying the operator combinations to different domains or considering various features of the query (vocabulary): While optimizing pre/post processing on VSM improves the performance of bug localization, different IR techniques might react differently to such processing. This can be a worthwhile candidate for future research direction. Additionally, expanding the vocabulary with acronym/abbreviation [50] from the bug report or checking the length of the query (vocabulary) and apply those as one of the features can be an interesting study to conduct in the future for the field of bug localization.

6. Related Work

Rao et al. [5] undertook a comparative study on the different IRBL techniques. The conclusion of their study showed that IRBL techniques are as effective as other fault localisation techniques. One particular conclusion of the study was that complicated models (such as LDA) do not outperform simpler models such as the VSM. Building up on this conclusion, DIGBUG employs the VSM to validate its approach. Our work, DIGBUG employs VSM as well by following its effectiveness from the literature [5, 26, 20].

Zhou et al. [26] proposed a revised VSM model that takes in additional parameters such as the different sizes of the source files and comparing the similarity between a new bug and previously fixed bugs. This revised model built on two hypotheses: (1) larger source files are more likely to contain bugs, and (2) a bug that was previously fixed can help locate the relevant files for a similar new bug. A tool called BugLocator was developed that demonstrated an outperforming result when compared to previous IRBL techniques.

Meanwhile, Hill et al. [51] undertook a study to investigate the impact of stemming within bug localization corresponding to different types of queries. Additionally, they compared different stemming algorithms, and their results proved that the efficiency corresponds to the length of the query. DIGBUG using stemming (STM) as one of the Preprocessing operators in its bug localization approach.

Later, Kim et al. [52] observed that many bugs reports do not have sufficient information that is necessary to make

a good prediction, which ultimately makes bug localization unsuccessful. To address this issue, they mark such bug reports as unpredictable and discard them to improve localization performance. Furthermore, they proposed a two-phase model that first checks the localizability and only considers them predictable. The results show that using the predictive two-phase approach, on average, a 70% likelihood.

Saha el at. [6] implemented BLUiR and improved the results of the BugLocator by incorporating the structural information from the project source code. Their main observation was that source files are structured documents, and code constructs such as the class name and method names can improve localization for bug reports. The effectiveness of this technique was evaluated against the BugLocator and showed a significant improvement. Later Saha et al. [53] investigated whether IRBL approaches can work with source code that is written in a programming language that is not based on the concept of object-oriented programming. They found that although IRBL was effective for such source code, the integration of program structure information did not help provide better results.

Thomas et al. [20] investigated the consideration to be taken in the configuration of the classifier. Their study includes understand the use of general parameters (e.g., "Bug report representation", "Entity representation", "Pre-processing steps") and specific parameters for each of the technique (e.g., "Term weight", "Similarity metric", "Number of topics"). The experimental results demonstrated that the configuration of the IR-based classifier matters and one configuration (i.e., the best one in the literature) improves results in almost all cases. According to the results of our study, we provide insight that; the 'one-configuration-fits-all' strategy is not appropriate for bug localization due to the variety of attributes of bug reports.

Wang and Lo [54] decided to improve the precision performance by combining multiple existing information (i.e., version history, similar bug reports, structural information). Their tool, AmaLgam, only considers very recent version history, uses the same bug prediction technique as Google (i.e., BugLocator). Unlike Sisman and Kak [55], they assigned weights of contributions of each file by integrating BugLocator and BLUiR. The proposed approach achieved 12% to 16% improvements against BugLocator and BLUiR in terms of MAP.

While various information on bug reports and source code were being considered, Wong et al. [7] observed that stack traces from the input bug reports are important elements to consider. They divided source code into segments based on their similarity and analyzed the stack traces to localize the correct files to fix. They compared their results against BugLocator that showed to outperform it. They also discovered and reported that segmentation and stack trace analysis complement each other for boosting performance.

Similarly, Lobster [56] employs stack traces by combining textual similarity between a bug report, a code element, and the structural similarity between the stack trace with the code elements.

To accurately rank the source code file, Ye et al. [13] leveraged six features (i.e., Lexical Similarity, Collaborative Filtering Score, Class Name Similarity, Bug-Fixing Recency, Bug-Fixing Frequency, and Feature Scaling). The ranking score of each source file is calculated as a weighted combination of former features that are incorporating domain knowledge. At the same time, the weights are trained previously by using the learning-to-rank technique. They undertook the evaluation against the baselines (i.e., VSM and Usual Suspect) and the state-of-the-art (i.e.., BugLocator, BugScout). The results showed correct recommendations within the top 10 ranked files for over 70% of the bug reports in the Eclipse and Tomcat data, showing considerable improvements.

Some researchers continued investigations towards refining the complicated models. In the same, Youm et al. [28] proposed a combinational approach named BLIA, that statically integrates the analytics approach by using texts and stack traces from bug reports, AST, and code change histories. However, the results of BLIA showed that it could not provide consistent results in ranking the correct files compared to the state-of-the-art; MAP and MRR values are higher than them for some subjects.

Wang et al. [57] undertook an empirical study to evaluate the needs and the usefulness of IRBL. The study was based on analytical investigation supplemented by a user study. The results from the analytical study provided insights in regards to the valuable information that is often missing from bug reports that are needed for bug localization. Furthermore, developers can be guided automatically to the target source code files when high-quality bug reports are provided. This implied the marginal impact of the IRBL. Although IRBL has limited benefits yet, providing the list of suspicious files may still help developers get to the correct files faster. The user study results showed the fundamental importance of user studies on the techniques to get practical insights.

Another empirical study [57] pinpointed that in some cases, current IRBL techniques do not help developers in improving the bug localization. To overcome this weakness, Wen et al. [27] proposed Locus, which offers finer granularity than file-level and provides important contextual clues for localization. Unlike existing techniques, Locus retrieves information from software changes instead of source code tokens. Maximum 20.5% points respectively improve the MAP and MRR on average comparing against the state-of-the-art (e.g., BRTracer [7], BLUiR [6], and AmaLgam). Locus also successfully located the bug-inducing changes within the top 5 for 41.0% of the bugs.

Wang and Lo [9] proposed AmaLgam+, which collects and cares for five sources of information (i.e., version history, similar reports, structure, stack traces, and reporter information) and integrates them with a composer. AmaLgam+ outperforms with 4% improvement on average than a prequel, AmaLgam, which outperforms the state-of-the-art. Similarly, DIGBUG considers information from attributes such as stack trace and the type of reporter from the given bug report along with other attributes such as code entities, de-

scriptions, and attachments.

Panichella et al. [19] deal with studying different approaches for removing special characters, identifier splitting, stop word removal, stemming, term weighting, IR engine, and similarity measure on bug localization. As a result, they devised an approach towards finding the best possible operators using genetic algorithms. Instead of our tool DIGBUG, they worked on grouping bug reports by different configurations rather than investigating the best operator combination for each characteristic of bug reports.

Rahman and Roy [15] introduce BLIZZARD, which predicts buggy files by going through a query (i.e., bug report) reformulation technique. It checks whether there are excessive program entities (i.e., stack trace, fully natural language token-based, program elements) or not in the incoming bug reports, then applies different reformulations. Moreover, it leverages TextRank (a graph-based term weighting method) for developing the Trace Graphs instead of TF-IDF to identify important keywords. The localization results report that BLIZZARD can outperform the state-of-the-art, and it also improves 22% and 20% more of noisy queries and poor queries, respectively, others.

Many researchers have applied deep learning techniques to bug location problems. Huo et al. [58] adopted the use of the pairwise learning-to-rank approach to classifying the bug reports and source code files into linked and non-linked records. They proposed a new architecture (called Natural Language And Programming Language CNN) that outperformed the other state-of-the-art bug localization models. Later, Huo et al. [59] proposed using an extension to the model called TRANP-CNN that leveraged cross-project information for bug localization. The model works by first extracting features from bug reports and source code files of source and target projects, then generated project-specific predictions for new bugs by the extracted features. These features allowed a higher accurate matching of the source-code files with the new bugs for localization. Huo et al. [60] also proposed another new architecture for the bug localization problem using a combination of LSTM and CNN called LS-CNN. The LS-CNN exploits the sequential nature of the source code such as functional semantics of program and the correlation between bug report and source code that identifies the buggy files. LS-CNN combines LSTM and CNN for the processing, where LSTM focuses on the extraction of semantic features and handling dependencies between different source code statements while the CNN captures the local and structural information within statements. They compared their proposed architecture against NP-CNN, CNN, LSTM, and other state-of-the-art bug localization approaches such as BugLocator [26] and HyLoc [61].

Researchers conduct similar approach (i.e., applying various configurations) on other domains. Monero et al [62] investigated the effects of various text retrieval configurations and proposed an approach named QUEST to perform identification of artifacts based on the configuration that is most suitable for a given query. Also, Mills et al. [63] evaluated the application of automatic query quality prediction on software artifacts. They found that their approach can predict the results by evaluating the text in queries that increased efficiency in time manner and efforts of the search.

7. Conclusion

Bug localization is an expensive and difficult task, especially for large software projects. However, the benefits of an efficient bug localization technique can improve the manner in which developers handle and address the bugs in their projects. In reality, use of bug localization techniques remains far from being used due to the issues.

In this paper, we proposed our key insight that it is more viable to consider the attributes from the bug reports for an IR-based bug localization technique. We present DIGBUG, which builds on this insight by leveraging the attributes extracted from the bug reports to figure out the best-performing pre/post-processing operators. Furthermore, we demonstrated how the performance of bug localization is improved by DIGBUG. This was evaluated on the dataset from Bench4BL [18] that allows us to compare the results against state-of-the-art techniques. The results were evaluated on a total of 9,459 bug reports as input for DIGBUG and showed an improvement of 6 and 14 percentage points in MAP and MRR, respectively.

Although our work shows a significant gain over the state-of-the-art techniques, the results can be improved. For example, it can apply different types pre/post-processing operators and/or attributes from bug reports. In addition, DIGBUG can be plugged into other IRBL techniques.

We provide a replication package with datasets and scripts as DIGBUG, at `https://github.com/FalconLK/DigBug-Dig-into-Bug`.

Acknowledgements

This work was supported by the NATURAL project, which has received funding from the European Research Council under the European Union's Horizon 2020 research and innovation programme under grant agreement No 949014, Fonds National de la Recherche (FNR), Luxembourg, under FNR-AFR PhD/11623818, the National Research Foundation of Korea (NRF) grant funded by the Korea government (MSIT) (No. 2021R1A5A1021944 and 2021R1A5A1021944), the National Natural Science Foundation of China (Grant No. 62172214), the Natural Science Foundation of Jiangsu Province, China (Grant No. BK20210279), and the National Research Foundation, Singapore, under its Industry Alignment Fund – Pre-positioning (IAF-PP) Funding Initiative. Any opinions, findings and conclusions or recommendations expressed in this material are those of the author(s) and do not reflect the views of National Research Foundation, Singapore. Additionally, the research was supported by Kyungpook National University Research Fund, 2020.

References

[1] Hive-issues, https://issues.apache.org/jira/projects/HIVE/ issues/ (Last Accessed: Oct. 2020).

[2] X. Huo, F. Thung, M. Li, D. Lo, S.-T. Shi, Deep transfer bug localization, IEEE Transactions on Software Engineering (2019) 1–1Conference Name: IEEE Transactions on Software Engineering.

[3] G. Gay, S. Haiduc, A. Marcus, T. Menzies, On the use of relevance feedback in ir-based concept location, in: 2009 IEEE International Conference on Software Maintenance, IEEE, 2009, pp. 351–360.

[4] S. K. Lukins, N. A. Kraft, L. H. Etzkorn, Bug localization using latent dirichlet allocation, Information and Software Technology 52 (9) (2010) 972–990.

[5] S. Rao, A. 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, ACM, 2011, pp. 43–52.

[6] R. K. Saha, M. Lease, S. Khurshid, D. E. Perry, Improving bug localization using structured information retrieval, in: 2013 28th IEEE/ACM International Conference on Automated Software Engineering (ASE), 2013, pp. 345–355.

[7] C.-P. Wong, Y. Xiong, H. Zhang, D. Hao, L. Zhang, H. Mei, Boosting bug-report-oriented fault localization with segmentation and stack-trace analysis, in: 2014 IEEE International Conference on Software Maintenance and Evolution, 2014, pp. 181–190, ISSN: 1063-6773.

[8] A. Koyuncu, T. F. Bissyandé, D. Kim, K. Liu, J. Klein, M. Monperrus, Y. L. Traon, D&c: A divide-and-conquer approach to IR-based bug localization, arXiv:1902.02703 [cs] (2019).

[9] S. Wang, D. Lo, AmaLgam+: Composing rich information sources for accurate bug localization, Journal of Software: Evolution and Process 28 (10) (2016) 921–942.

[10] K. C. Youm, J. Ahn, E. Lee, Improved bug localization based on code change histories and bug reports, Information and Software Technology 82 (2017) 177–192.

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

[12] A. Schroter, A. Schröter, N. Bettenburg, R. Premraj, Do stack traces help developers fix bugs?, in: 2010 7th IEEE Working Conference on Mining Software Repositories (MSR 2010), IEEE, 2010, pp. 118–121.

[13] X. Ye, R. Bunescu, C. Liu, Learning to rank relevant files for bug reports using domain knowledge, in: Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering, FSE 2014, Association for Computing Machinery, 2014, pp. 689–699.

[14] C. Tantithamthavorn, S. L. Abebe, A. E. Hassan, A. Ihara, K. Matsumoto, The impact of ir-based classifier configuration on the performance and the effort of method-level bug localization, Information and Software Technology 102 (2018) 160–174.

[15] M. M. Rahman, C. 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, ESEC/FSE 2018, ACM, 2018, pp. 621–632.

[16] P. Loyola, K. Gajananan, F. Satoh, Bug localization by learning to rank and represent bug inducing changes, in: Proceedings of the 27th ACM International Conference on Information and Knowledge Management, ACM, 2018, pp. 657–665.

[17] G. Liu, Y. Lu, K. Shi, J. Chang, X. Wei, Mapping bug reports to relevant source code files based on the vector space model and word embedding, IEEE Access 7 (2019) 78870–78881.

[18] J. Lee, D. Kim, T. F. Bissyandé, W. Jung, Y. Le Traon, Bench4bl: Reproducibility study on the performance of IR-based bug localization, in: Proceedings of the 27th ACM SIGSOFT International Symposium on Software Testing and Analysis, ISSTA 2018, ACM, 2018, pp. 61–72.

[19] A. Panichella, B. Dit, R. Oliveto, M. D. Penta, D. Poshyvanyk, A. D. Lucia, Parameterizing and assembling IR-based solutions for SE tasks using genetic algorithms, in: 2016 IEEE 23rd International Conference on Software Analysis, Evolution, and Reengineering (SANER), Vol. 1, 2016, pp. 314–325.

[20] S. W. Thomas, M. Nagappan, D. Blostein, A. E. Hassan, The impact of classifier configuration and classifier combination on bug localization, IEEE Transactions on Software Engineering 39 (10) (2013) 1427–1443.

[21] D. Binkley, D. Lawrie, C. Uehlinger, D. Heinz, Enabling improved IR-based feature location, Journal of Systems and Software 101 (2015) 30–42.

[22] R. Abreu, P. Zoeteweij, A. van Gemund, On the Accuracy of Spectrum-based Fault Localization, in: Testing: Academic and Industrial Conference Practice and Research Techniques - MUTATION, 2007. TAICPART-MUTATION 2007, 2007, pp. 89 –98.

[23] R. Abreu, P. Zoeteweij, A. J. C. van Gemund, An Evaluation of Similarity Coefficients for Software Fault Localization, in: 12th Pacific Rim International Symposium on Dependable Computing, 2006. PRDC '06, IEEE, 2006, pp. 39–46.

[24] N. DiGiuseppe, J. A. Jones, Fault density, fault types, and spectra-based fault localization, Empirical Software Engineering 20 (4) (2014) 928–967.

[25] W. E. Wong, R. Gao, Y. Li, R. Abreu, F. Wotawa, A Survey on Software Fault Localization, IEEE Transactions on Software Engineering 42 (8) (2016) 707–740.

[26] J. Zhou, H. Zhang, D. Lo, Where should the bugs be fixed? - more accurate information retrieval-based bug localization based on bug reports, in: Proceedings of the 2012 International Conference on Software Engineering, IEEE Press, Piscataway, NJ, USA, 2012, pp. 14–24.

[27] M. Wen, R. Wu, S.-C. Cheung, Locus: locating bugs from software changes, in: Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering, ACM, Singapore, Singapore, 2016, pp. 262–273.

[28] K. C. Youm, J. Ahn, J. Kim, E. Lee, Bug localization based on code change histories and bug reports, in: 2015 Asia-Pacific Software Engineering Conference (APSEC), 2015, pp. 190–197.

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

[30] D. M. Blei, A. Y. Ng, M. I. Jordan, Latent dirichlet allocation, Journal of machine Learning research 3 (Jan) (2003) 993–1022.

[31] G. Salton, A. Wong, C. S. Yang, A Vector Space Model for Automatic Indexing, Communications of the ACM 18 (11) (1975) 613–620.

[32] S. T. Dumais, G. W. Furnas, T. K. Landauer, S. Deerwester, R. Harshman, Using latent semantic analysis to improve access to textual information, in: Proceedings of the SIGCHI Conference on Human Factors in Computing Systems, CHI '88, Association for Computing Machinery, 1988, pp. 281–285.

[33] A. K. Jain, M. N. Murty, P. J. Flynn, Data clustering: a review, ACM computing surveys (CSUR) 31 (3) (1999) 264–323.

[34] A. Koyuncu, K. Liu, T. F. Bissyandé, D. Kim, M. Monperrus, J. Klein, Y. Le Traon, iFixR: Bug report driven program repair, in: Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering, ESEC/FSE 2019, ACM, 2019, pp. 314–325.

[35] B. Dit, M. Revelle, M. Gethers, D. Poshyvanyk, Feature location in source code: a taxonomy and survey, Journal of Software: Evolution and Process 25 (1) (2013) 53–95.

[36] W. Y. Chong, B. Selvaretnam, L.-K. Soon, Natural language processing for sentiment analysis: an exploratory analysis on tweets, in: 2014 4th International Conference on Artificial Intelligence with Applications in Engineering and Technology, IEEE, 2014, pp. 212–217.

[37] F. Sun, A. Belatreche, S. Coleman, T. M. McGinnity, Y. Li, Preprocessing online financial text for sentiment classification: A natural language processing approach, in: 2014 IEEE Conference on Computational Intelligence for Financial Engineering & Economics (CIFEr), IEEE, 2014, pp. 122–129.

[38] P. S. Kochhar, Y. Tian, D. Lo, Potential biases in bug localization: Do they matter?, in: Proceedings of the 29th ACM/IEEE interna-

tional conference on Automated software engineering, ACM, 2014, pp. 803–814.
[39] S. Rahman, M. M. Rahman, K. Sakib, A statement level bug localization technique using statement dependency graph., in: ENASE, 2017, pp. 171–178.
[40] D. Kılınç, F. Yücalar, E. Borandağ, E. Aslan, Multi-level reranking approach for bug localization, Expert Systems 33 (3) (2016) 286–294.
[41] M. Rath, D. Lo, P. Mäder, Analyzing requirements and traceability information to improve bug localization, in: 2018 IEEE/ACM 15th International Conference on Mining Software Repositories (MSR), IEEE, 2018, pp. 442–453.
[42] Y. Tian, D. Wijedasa, D. Lo, C. Le Goues, Learning to rank for bug report assignee recommendation, in: 2016 IEEE 24th International Conference on Program Comprehension (ICPC), IEEE, 2016, pp. 1–10.
[43] B. Qi, H. Sun, W. Yuan, H. Zhang, X. Meng, Dreamloc: A deep relevance matching-based framework for bug localization, IEEE Transactions on Reliability (2021) 1–15doi:10.1109/TR.2021.3104728.
[44] N. Bettenburg, R. Premraj, T. Zimmermann, S. Kim, Extracting structural information from bug reports, in: Proceedings of the 2008 international working conference on Mining software repositories, ACM, 2008, pp. 27–30.
[45] Wicket-issues, https://issues.apache.org/jira/projects/WICKET/issues/ (Last Accessed: Oct. 2020).
[46] D. Project, https://code.djangoproject.com/ticket/24117 (July. 2013).
[47] H. B. Mann, On a Test of Whether one of Two Random Variables is Stochastically Larger than the Other, The Annals of Mathematical Statistics 18 (1) (1947) 50–60.
[48] Y. Xiao, J. Keung, K. E. Bennin, Q. Mi, Improving bug localization with word embedding and enhanced convolutional neural networks, Information and Software Technology 105 (2019) 17–29.
[49] G. Antoniol, K. Ayari, M. Di Penta, F. Khomh, Y.-G. Guéhéneuc, Is it a bug or an enhancement? a text-based approach to classify change requests, in: Proceedings of the 2008 conference of the center for advanced studies on collaborative research: meeting of minds, CASCON '08, Association for Computing Machinery, 2008, pp. 304–318.
[50] D. Binkley, D. Lawrie, C. Uehlinger, Vocabulary normalization improves ir-based concept location, in: 2012 28th IEEE International Conference on Software Maintenance (ICSM), IEEE, 2012, pp. 588–591.
[51] E. Hill, S. Rao, A. Kak, On the use of stemming for concern location and bug localization in java, in: 2012 IEEE 12th International Working Conference on Source Code Analysis and Manipulation, 2012, pp. 184–193.
[52] D. Kim, Y. Tao, S. Kim, A. Zeller, Where should we fix this bug? a two-phase recommendation model, IEEE transactions on software Engineering 39 (11) (2013) 1597–1610.
[53] R. K. Saha, J. Lawall, S. Khurshid, D. E. Perry, On the effectiveness of information retrieval based bug localization for c programs, in: 2014 IEEE International Conference on Software Maintenance and Evolution, 2014, pp. 161–170, ISSN: 1063-6773.
[54] S. Wang, D. Lo, Version history, similar report, and structure: Putting them together for improved bug localization, in: Proceedings of the 22Nd International Conference on Program Comprehension, ICPC 2014, ACM, 2014, pp. 53–63.
[55] B. Sisman, A. C. Kak, Incorporating version histories in information retrieval based bug localization, in: 2012 9th IEEE Working Conference on Mining Software Repositories (MSR), 2012, pp. 50–59, ISSN: 2160-1860.
[56] L. Moreno, J. J. Treadway, A. Marcus, W. Shen, On the use of stack traces to improve text retrieval-based bug localization, in: 2014 IEEE International Conference on Software Maintenance and Evolution, 2014, pp. 151–160.
[57] Q. Wang, C. Parnin, A. Orso, Evaluating the usefulness of IR-based fault localization techniques, in: Proceedings of the 2015 International Symposium on Software Testing and Analysis, ISSTA 2015, Association for Computing Machinery, 2015, pp. 1–11.
[58] X. Huo, M. Li, Z.-H. Zhou, et al., Learning unified features from natural and programming languages for locating buggy source code., in: IJCAI, Vol. 16, 2016, pp. 1606–1612.
[59] X. Huo, F. Thung, M. Li, D. Lo, S.-T. Shi, Deep transfer bug localization, IEEE Transactions on software engineering (2019).
[60] X. Huo, M. Li, Enhancing the unified features to locate buggy files by exploiting the sequential nature of source code., in: IJCAI, 2017, pp. 1909–1915.
[61] A. N. Lam, A. T. Nguyen, H. A. Nguyen, T. N. Nguyen, Combining deep learning with information retrieval to localize buggy files for bug reports (n), in: 2015 30th IEEE/ACM International Conference on Automated Software Engineering (ASE), IEEE, 2015, pp. 476–481.
[62] L. Moreno, G. Bavota, S. Haiduc, M. Di Penta, R. Oliveto, B. Russo, A. Marcus, Query-based configuration of text retrieval solutions for software engineering tasks, in: Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering, 2015, pp. 567–578.
[63] C. Mills, G. Bavota, S. Haiduc, R. Oliveto, A. Marcus, A. D. Lucia, Predicting query quality for applications of text retrieval to software engineering tasks, ACM Transactions on Software Engineering and Methodology (TOSEM) 26 (1) (2017) 1–45.