2022 IEEE Conference on Software Testing, Verification and Validation (ICST)

An Empirical Study of IR-based Bug Localization for Deep Learning-based Software

Misoo Kim
Institute of Software Convergence, Sungkyunkwan University
Suwon, Republic of Korea
misoo12@skku.edu

Youngkyoung Kim
Department of Electrical and Computer Engineering, Sungkyunkwan University
Suwon, Republic of Korea
agnes66@skku.edu

Eunseok Lee[*]
College of Computing and Informatics, Sungkyunkwan University
Suwon, Republic of Korea
leees@skku.edu

***Abstract*—As the impact of deep-learning-based software (DLSW) increases, automatic debugging techniques for guaranteeing DLSW quality are becoming increasingly important. Information-retrieval-based bug localization (IRBL) techniques can aid in debugging by automatically localizing buggy entities (files and functions). The low-cost advantage of IRBL can alleviate the difficulty of identifying bug locations due to the complexity of DLSW. However, there are significant differences between DLSW and traditional software, and these differences lead to differences in search space and query quality for IRBL. That is, IRBL performance must be validated in DLSW.**

We empirically validated IRBL performance for DLSW from the following four perspectives: 1) similarity model, 2) query generation, 3) ranking model for buggy file localization, and 4) ranking model for buggy function localization. Based on four research questions and a large-scale experiment using 2,365 bug reports from 136 DLSW projects, we confirmed the salient characteristics of DLSW from the perspective of IRBL and derived four recommendations for practical IRBL usage in DLSW from the empirical results. Regarding IRBL performance, we validated that IRBL performance with the combination of bug-related features outperformed that of using only file similarity by 15% and IRBL ranked buggy files and functions on average of 1.6th and 2.9th, respectively. Our study is valuable as a baseline for IRBL researchers and as a guideline for DLSW developers who wish to apply IRBL to ensure DLSW quality.

***Index Terms*—Empirical study, Deep learning-related software, Information retrieval-based bug localization, Python bugs**

I. Introduction

As the popularity of deep-learning-related software (DLSW) and its impact on society have increased, ensuring the quality of DLSW has become increasingly critical [1]. Automatic debugging technology is required to guarantee the quality of DLSW. However, automatic debugging is difficult because of the complexity and black-box characteristics of DLSW [2]. Therefore, it is necessary to study bugs and debugging technologies for DLSW. Existing DLSW-related studies have focused on deriving new insights by investigating bugs in DLSW [3]–[6] or proposing debugging techniques in the deep learning (DL) models, not entire software [7], [8]. However, there is still a lack of interest in the debugging technology for DLSW from the perspective of software (SW), including not only DL models but also various other functionalities.

[*] Corresponding author

To debug DLSW from the software perspective, it is necessary to develop an automatic bug localization technique based on bug reports for localizing bugs in both the DL model and other functions. The first step of debugging is bug localization. Bug reports provide vital information for bug localization [9]. Similar to general software, in DLSW, bugs are reported through bug reports and it takes an average of 30 days to resolve a reported bug [10]. The information-retrieval-based bug localization (IRBL) technique is a representative technique that identifies buggy files and functions (methods) by using bug reports as queries and setting source files or functions as a search space [11]. This technique outputs a ranked list of files or functions according to buggy suspiciousness. Developers can resolve bugs by examining the source files and functions in order of the ranked list [12]. This technique offers an advantage in cost over the spectrum-based technique, the most popular bug localization technique. The spectrum-based technique requires software testing, while IRBL does not [13].

The high complexity of DLSW can draw DLSW developer's attention to IRBL due to the following. The low-cost advantage of IRBL can alleviate the difficulty of identifying bug locations due to the complexity of DLSW. In addition, DLSW developers require more exact and valuable information for bug resolution because the complexity of DLSW is known to be higher than that of general SW [2], which leads bug reporters to provide more information in the bug reports for DLSW. That is, the quality of the bug reports in DLSW is likely to be higher than that in traditional software. General belief is that the performance of IRBL depends on the quality of bug reports [14]. DLSW developers might expect higher IRBL performance while this cannot be ensured before running and validating IRBL. Therefore, the IRBL performance needs to be validated within each software domain.

The performance of IRBL has mainly been evaluated and reported for general SW and there has been little focus on DLSW. However, there are some apparent differences between traditional SW and DLSW. Saleema et al. and Han et al. explained that DLSW has a different development process and workflow than traditional SW [2], [15]. Jebnoun et al. demonstrated that DLSW is more complex than general SW [6]. Zhang et al. and Humbatova et al. presented DL-specific bugs that are clearly distinguished from bugs in general SW [3], [16]. The following two differences also distinguish DLSW from the perspective of IRBL.

The first difference is the **characteristic coding style** of DLSW. There are many application programming interface calls and glue codes in DLSW than the general SW [17]. These characteristics lead to distinctions of the textual characteristics of source files and functions in DLSW from traditional SW. The second difference is the **valued information in the bug reports** available for resolving bug reports. For example, researchers have noted that environment information is less important than other information in the investigation results for general SW [18]. However, as the DL-related frameworks and libraries used in most DLSW development continue to evolve rapidly [5], [19], the environment in which a reporter observes a bug is becoming critical information for developers to reproduce DLSW bugs [20]. In other words, because developers of DLSW may value environmental information more highly than general SW developers, there is a high probability that this information will be requested from bug reporters. This scenario results in different characteristics of DLSW bug reports compared to general SW bug reports from the perspective of IRBL.

These two differences make it difficult to guarantee and trust that the previously reported performance in IRBL studies will be consistent for DLSW. In other words, the validation of IRBL performance on DLSW is required for DLSW developers. The IRBL technique consists of textual similarity analysis between bug reports and retrieval targets (e.g., source files or functions) [21], [22], query generation [23], [24], and the ranking of retrieved targets based on the weighted summation of similarity and various bug-related features [11], [25]–[28]. Traditional IRBL studies focused on localizing buggy files and few recent studies have focused on localizing buggy functions (methods) [29], [30]. For debugging efficiency, localizing buggy functions is better than localizing buggy files for developers [12]. In this study, we validated the performance of IRBL in DLSW for the three components outlined above for both buggy file- and buggy function-localization. We established the following four research questions (RQs), each answered based on our experimental results using 2,365 bug reports from 136 DLSW projects in Denchmark, a recent bug benchmark for DLSW [10].

1) **RQ1. Which similarity model can localize buggy files effectively?** Through an experiment, we determined that one particular model exhibits a 22% higher mean average precision (MAP) compared with the other models.
2) **RQ2. Are all texts in a bug report effective as queries?** We classified texts into four types and separated them into 18 categories. Our experimental results demonstrated that environmental information, and headings represent noisy text in queries.
3) **RQ3. Which weights for bug-related features are the best for achieving effective IRBL?** We integrated five bug-related features using a weighted sum. On average, the best weights for bug report similarity, commit history, stack trace, hunk similarity, and source file similarity were 0.0, 0.2, 0.4, 0.2, and 0.2, respectively. With these optimal weights, IRBL outperforms the source file similarity alone with an MAP improvement of 15%.
4) **RQ4. How much performance can be expected from IRBL for buggy function localization?** The superior way to localize buggy functions for a target bug report is to use the summation of 1) source file scores computed by the weighted sum of bug-related features and 2) the similarity between the target bug report and functions. Experimental results demonstrate that this method can localize buggy functions in 2.9 ranks.

The answers to these questions can provide a foundation for DLSW developers to achieve practical IRBL usage. Our study is geared toward not only DLSW developers but also researchers studying IRBL on DLSW, as we are publishing both our experimental dataset and packages. The main contributions of this study can be summarized as follows:

- Our study is the first to apply and validate IRBL in DLSW. The results and methods of this study can be used as baselines for IRBL researchers focusing on DLSW.
- Our empirical study covered all the key elements related to IRBL in DLSW. We analyzed document similarity models, query effectiveness, and bug-related features. We also evaluated performance for buggy function localization based on IRBL, which has received little attention in the IRBL field.
- Large-scale experimental results provide guidelines for IRBL not only for researchers, but also for DLSW developers. Therefore, we can expect to resolve bugs quickly and improve the quality of DLSW by using IRBL.
- By providing both dataset creation script and IRBL tools, we contribute to future researchers and developers.

The remainder of this paper is organized as follows. Section II discusses the background of bug reports and IRBL. Section III discusses the study preparation phase. Sections IV, V, VI and VII describe the methods and results for answering the four research questions. Section VIII discusses our findings and threats to the validity of our results. Finally, Section IX discusses previous related studies and Section X presents our concluding remarks and future work.

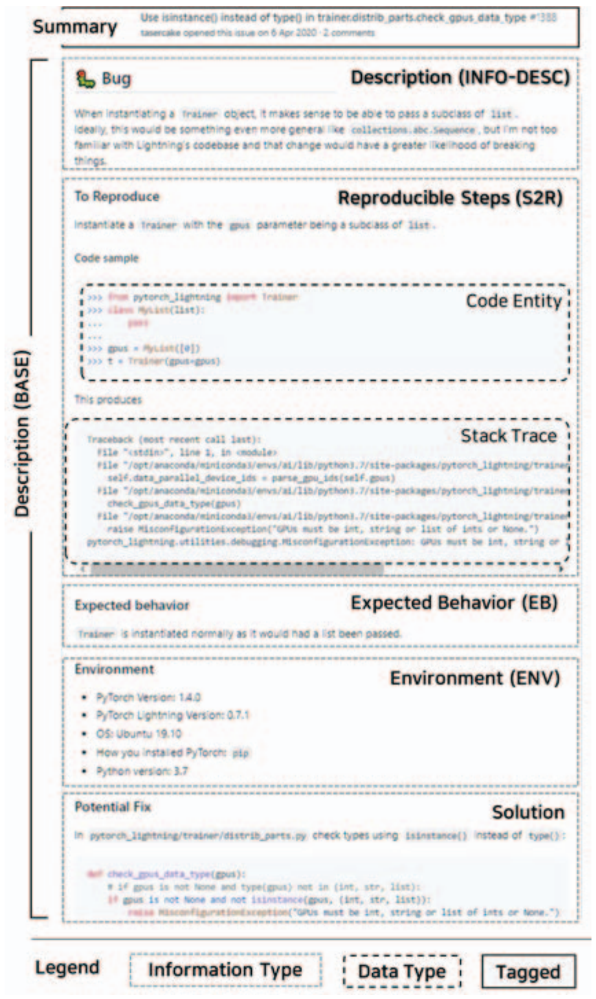
II. Background

A. Bug Report

Figure 1 presents an example bug report[1] for DLSW from GitHub. Bug reports can generally be divided into summary and description sections, as shown on the left of Figure 1. A reporter writes detailed information in the description area and summarizes the bug information in the summary. The queries for general IRBL are texts that incorporate summaries and descriptions [31].

Rahman et al. classified bug reports as having a stack trace, having a code entity, or having neither [23]. To classify bug reports, they first extracted the stack trace and code entity

[1] https://github.com/pytorchlightning/pytorch-lightning/issues/1388



Use isinstance() instead of type() in trainer.distrib_parts.check_gpus_data_type #1388
tasercake opened this issue on 6 Apr 2020 · 2 comments

Bug

When instantiating a `Trainer` object, it makes sense to be able to pass a subclass of `list`. Ideally, this would be something even more general like `collections.abc.Sequence`, but I'm not too familiar with Lightning's codebase and that change would have a greater likelihood of breaking things.

To Reproduce

Instantiate a `Trainer` with the `gpus` parameter being a subclass of `list`.

Code sample

```
>>> from pytorch_lightning import Trainer
>>> class MyList(list):
...     pass
...
>>> gpus = MyList([0])
>>> t = Trainer(gpus=gpus)
```

This produces

```
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "/opt/anaconda/miniconda3/envs/ai/lib/python3.7/site-packages/pytorch_lightning/trainer
    self.data_parallel_device_ids = parse_gpu_ids(self.gpus)
  File "/opt/anaconda/miniconda3/envs/ai/lib/python3.7/site-packages/pytorch_lightning/trainer
    check_gpus_data_type(gpus)
  File "/opt/anaconda/miniconda3/envs/ai/lib/python3.7/site-packages/pytorch_lightning/trainer
    raise MisconfigurationException("GPUs must be int, string or list of ints or None.")
pytorch_lightning.utilities.debugging.MisconfigurationException: GPUs must be int, string or
```

Expected behavior

`Trainer` is instantiated normally as it would had a list been passed.

Environment

- PyTorch Version: 1.4.0
- PyTorch Lightning Version: 0.7.1
- OS: Ubuntu 19.10
- How you installed PyTorch: `pip`
- Python version: 3.7

Potential Fix

In `pytorch_lightning/trainer/distrib_parts.py` check types using `isinstance()` instead of `type()`:

```
def check_gpus_data_type(gpus):
    # if gpus is not None and type(gpus) not in (int, str, list):
    if gpus is not None and not isinstance(gpus, (int, str, list)):
        raise MisconfigurationException("GPUs must be int, string or list of ints or None.")
```

Fig. 1: Example bug report

from the entire text of a bug report using regular expressions. In Figure 1, the bug report contains the stack trace and code entity (see the dotted boxes in the legend indicating "Data Type"). The stack trace and code entity can be used as textual and structural data, respectively.

Because a web-based bug reporting system is typically used, a pure bug report contains HTML tags. The tag <h> indicates the header of the content [32]. Texts from each heading in a bug report to the next heading can be considered as text elements that explain the information that each heading contains. In Figure 1, one can identify five pieces of information by heading: description (INFO-DESC), steps to reproduce the bug (S2R), expected behavior (EB), environment (ENV), and solution. In addition to the heading tag, the <code> tag is used to highlight source code [33]–[35]. In the example, there are texts wrapped with this tag in light gray boxes. Therefore, the provided text types can be inferred from HTML tags.

As described above, bug reports are a mixture of various types of information and data. Basic IRBL uses the full text of a bug report as a query without this distinction. Some studies have demonstrated that query performance can change depending on the information described by a reporter or structured texts provided in a bug report [23]. Our study separated texts in the bug report based on 1) regular expressions, 2) heading, and 3) HTML tags.Based on this separation, we analyzed whether all textual features are suitable for use as the query.

B. IRBL

The IRBL technique defines a set of source files (or functions) as a search target and a bug report as a query, and computes buggy suspicion scores for each file. The most commonly used score is the textual similarity between bug reports and source files [36]. Additionally, existing IRBL tools extract bug-related features, compute suspicious scores based on these features, and rank the source files based on these scores [28].

This section describes text preprocessing and two core concepts for IRBL, namely textual similarity models and bug-related features, and evaluation metrics for IRBL.

1) Text preprocessing: The first step in computing document similarity is text preprocessing. Preprocessing consists of text normalization and stop-word removal [23]. Text normalization involves the removal of punctuation marks, tokenization, and splitting of identifiers. This step divides words into meaningful tokens using splitting algorithms such as the camel case and snake splitting algorithms. Stop-word removal removes unhelpful words such as "the." Preprocessing is equally applied to all bug-related features that require document similarity calculation.

2) Similarity models: The main models used for document similarity in the information retrieval field are the term-frequency-inversed-document-frequency (TFIDF)-based vector space model (VSM) and BM25 [36]. The similarity between a bug report and a source file (FileSim) is the basic component of IRBL. Zhang et al. proposed the revised VSM (rVSM) under the assumption that longer source files tend to be more buggy [11]. Our study determined which model performs best for localizing buggy files based on experiments with three similarity models.

3) Bug-related features: The features used by existing IRBL tools for bug localization are divided into five elements: similarity between a bug report and source file (FileSim), similarity between a bug report and past bug reports (BugSim), commit recency (Comm), stack trace (Strace), and similarity between a bug report and hunk (HunkSim). The similarity model for FileSim was described in the previous section, so we will describe the remaining four features in this section.

BugSim. The similarity to past bug reports is adopted based on the assumption that source files changed to resolve a similar bug report would also be buggy [11]. Equation 1 defines the metric used to compute this score.

$$BugSim(b, f) = \sum_{b' \in \{b' | b' \in B \cap f \in F_{b'}\}} \frac{sim(b, b')}{|b_f|} \qquad (1)$$

Here, b is the bug report used as a query, f is the source file, B is a set of past bug reports b', and $F_{b'}$ is a set of source files modified to solve past bug reports *b'*. This metric increases the score corresponding to the source file *f* that was modified in response to similar bug reports in the past.

Comm. The second feature is the commit recency score. This feature is based on the finding that recently modified source files are more likely to be buggy [27], [37]. This score is computed based on the modification date for each source file in the commit history before bugs were reported. This score is calculated using Equation 2.

$$Comm(f,k) = \sum_{c \in C \cap f \in c} \frac{1}{1 + e^{12(1-(\frac{k-d_c}{k}))}} \tag{2}$$

This metric analyzes how recently a commit *c* occurred relative to *k*. *C* is the set of past commits *c* and *f* is the source file modified by the past commit *c*. d_c is the number of elapsed days between the bug reporting time and commit time, and *k* is a parameter representing the number of days before commit occurs.

Strace. The third feature is the stack-trace-based score. This feature is based on the fact that source files located higher (lower in the case of Python traces) within the trace are more likely to be defective. This score was proposed for BRTracer [26]. This score is computed as shown in Equation 3.

$$Strace(b,f) = \begin{cases} \frac{1}{rank}, & f \in b_{strace} <= 10 \\ 0.1, & f \in b_{strace} > 10 \\ 0, & \text{otherwise.} \end{cases} \tag{3}$$

This equation scores the source files appearing in the top-10 trace files according to the inverse rank of each file. The other source files are scored as 0.1 if they are included in the stack trace. If source files are not included in the stack trace, their scores are set to zero.

HunkSim. The final feature is the similarity score between bug reports and hunks. This metric ranks source files linked to hunks with high similarity to bug reports. This score was proposed by Locus [38]. In our study, hunks contained code changes and a summary of commits. This score is computed using Equation 4.

$$HunkSim(b,f) = \max_{h \in H \cap f \in h} sim(b,h) \tag{4}$$

H is the set of past hunks *h* and *sim* is the similarity between a hunk and bug report *b*. After calculating the similarity to all hunks in the past, the largest value is assigned as a score to the source file.

In our study, we validated which bug-related features are suitable for IRBL in DLSW when IRBL is performed based on the four elements described in this section, along with FileSim, which is described in Section II-B2, by calculating weighted sums.

4) Evaluation metric: There are three standard evaluation metrics in IRBL: top-k rank ratio (Top-k), mean reciprocal rank (MRR), and mean average precision (MAP). Higher values of these computed metrics indicate better performance than lower values.

Top-k. This metric refers to the percentage of bug reports that localize at least one buggy source file within the k rank. It can be computed using Equation 5.

$$Top\text{-}k = \frac{\sum_{i=1}^{N} topK(i)}{N} \tag{5}$$

where *i* is an index of each bug report in the experimental dataset and *N* is the number of all bug reports. *topK(i)* is a binary indicator of whether the highest-ranked buggy source file in the *i*-th bug report is in the *k* rank.

MRR. This metric is the mean of reciprocal ranks. MRR represents the effort from a developer required to find the first buggy files. It can be computed using Equation 6.

$$MRR = \frac{1}{N} \sum_{i=1}^{N} \frac{1}{top\text{-}rank(i)} \tag{6}$$

where *i* is the index of each bug report, *N* is the number of bug reports, and *top-rank(i)* is the rank of the highest-ranked buggy file in the *i*-th bug report.

MAP. This metric evaluates the quality of all ranked buggy files. It represents the effort required from a developer to find all buggy files. MAP is calculated using Equation 7.

$$MAP = \frac{1}{N} \sum_{i=1}^{N} \frac{1}{M} \sum_{j=1}^{M} \frac{pos(j)}{j} \tag{7}$$

where *j* is the rank of each source file ranked from *i*-bug reports, *M* is the number of ranked files, and *pos(j)* is the number of buggy files among the *j-1* ranked files. *i* is the index of each bug report and *N* is the number of bug reports.

In addition, we conducted a statistical test using a Wilcoxon rank-sum test [39]. If the p-value is lower than 0.01, it indicates that the differences between the two data are significant. We noted this differences as *.

III. Study preparation

A. Research Questions

We defined the following four research questions to verify IRBL performance in DLSW and answered them based on experiments.

1) Which similarity model can localize buggy files the most effectively?
2) Are all texts in a bug report effective as a queries?
3) Which weights for bug-related features are optimal for achieving effective IRBL?
4) How much performance can be expected from IRBL for buggy function localization?

B. Experimental Dataset

1) Denchmark: Our empirical study was performed based on Denchmark, which is a recently published DLSW bug benchmark [10]. Denchmark provides 4,577 bug reports with three core HTML tags (<h>, <link> and <code>) for 193 DLSW projects. The ground-truth files and functions (methods) are also linked to each bug report. For IRBL, we must define the search space for each bug report for each

DLSW project. The following sections explain buggy version selection, bug report selection, and search space generation.

2) Buggy Version Selection: For bug localization, we must clearly define the search space of source files in each project. We applied multiple version matching to construct a file set of versions based on the reported date of each bug [40]. The most recently released version of the reporting date was selected as the search space for each bug report. If we can obtain buggy version information from bug reports, this information can help define a search space. However, bug reporters do not always provide version information. Assuming that a developer receives a bug report and localizes buggy files without knowing the buggy version, it is most likely that this task will be performed on the most recently released version, so we matched the buggy versions based on reported data.

3) Bug Report Selection: Because DLSW is typically implemented by Python [5], [41], we focused on bug reports that were resolved using only Python files. To evaluate IRBL performance for Python bug reports, the ground-truth files (or functions) had to be within the search space. We excluded bug reports in which at least one of the ground truth files (or functions) was not in the search space from our experiments.

4) Search Space Generation: We selected only the Python source files for the search space for IRBL because we focused on Python bug reports. We performed not only buggy file localization, but also buggy function localization. The search space for file localization was defined as the source files corresponding to buggy versions. Accordingly, all functions in the source files were extracted using the Python AST parser and defined as a search space for function localization.

5) Final Dataset: Table I presents the statistics of the final experimental dataset selected through the process described above.

TABLE I: Dataset Statistics

Loc	#P	#BRs	#Versions		#Entities		#BuggyEntities	
			Mean	Max	Mean	Max	Mean	Max
File	136	2,365	6.7	80	441.6	3,559	2.6	227
Method	130	1,817	6.1	63	5,048.1	58,070	3.9	638

P: Projects, BR: Bug reports

For buggy file localization, there were 2,365 bug reports for 136 projects. The average number of buggy versions was 6.7 and the maximum number was 80. The average number of source files was 441.6 and the maximum number was 3,559. The average number of buggy files in each bug report was 2.6 and the maximum number was 227.

For buggy function localization, there were 1,817 bug reports for 130 projects. The statistics of the buggy versions are identical to those of buggy file localization. The average number of functions was 5,048.1 and the maximum number of functions was 58,070. The average number of ground-truth functions was 3.9 and the maximum number was 638.

C. Bug Text Separation

To answer RQ2, we explain how to collect text from bug reports. The text in each bug report was divided into four types and 18 textual features were extracted. Table II summarizes and defines each feature and the number of bug reports associated with each feature.

TABLE II: Textual Features Summary

Type	Feature	#BR	Description
BASE	Sum	2,365	Title or summary of bug report
	Desc	2,365	Texts of bug report excluding summary
DATA	Trace	583	Stack trace or Back trace
	ErrMsg	1,056	Error or Exception Message
	CE	2,359	Source code entity
	NL_D	2,365	Bug report's text excluding the above texts
INFO	Bold	37	Not heading but highlighting texts
	Guide	61	Reporting guideline texts
	Solution	76	Solution proposed by reporters
	Extra	186	Additional information
	EB	225	Expected behavior or results
	OB	265	Observed behavior or results
	ENV	441	Environmental information
	S2R	562	Steps or code to reproduce bugs
	Desc	744	Bug description
TAG	Header	835	wrapped with <h>tags
	Link	947	wrapped with <a href>tags
	Code	1,174	wrapped with <code>tags
	NL_T	2,365	The texts not wrapped with the above tags

1) BASE features: The BASE type refers to the most basic textual features of bug reports. We first classify bug report summaries or titles as summary features (Sum) and classify the other texts as description features (Desc).

2) DATA-related features: The DATA type refers to structured textual data features. The texts in this type are divided into four elements: stack trace (Trace), error message (ErrMsg), code entity (CE), and other natural texts (NLD). BLIZZARD provides a set of regular expressions for extracting these features [23]. We obtained ErrMsg and CE using their tools. Because their tools are based on Java, we designed a regular expression to extract the trace from Python as follows.

```
Traceback.*\n(\s+.*\n)*(.*?)\n
```

We counted top-K from the bottom to compute Strace score in Equation 3, as stack trace of Python has a bottom-up priority. Then, the texts excluding ErrMsg, CE, and Strace are considered as NL_D.

3) INFO-related features: INFO-based textual features focus on the intended information in texts. There are nine textual features of the INFO type. These features were extracted based on heading labels. To this end, we first extracted heading texts that were wrapped by the <h> tag. We then manually labeled the texts by discussing the meaning of each heading. There are 521 headings. Among them, 86 headings are not exact heading but text highlighted by reporters (Bold). 41 are guideline texts on how to write bug reports (Guide). 37 indicate a potential solution with a cause (Solution). 73 represent additional information (Extra). 12 indicate expected results

(EB). 60 are observed results containing log and stack traces (OB). 67 indicate the environmental information of reporters (ENV). 78 headings explain bug reproduction steps and codes (S2R). 47 are general bug description parts (Desc). Based on the information associated with heading, we extracted the textual features belonging to the heading until the next heading appeared. The numbers of bug reports having each textual feature are reported in the third column of the table II.

4) TAG-based texts: The bug reports in Denchmark contain three core HTML tags (<h>, <link>, and <code>). Based on these HTML tags, we extracted three types of texts. The first is header text that is wrapped with <h> (header). The second is link information and related linked texts, which are attributes in <link> and wrapped with <link> (Link). The third is code information, as shown in Figure 1, which is wrapped by <code> (Code). We also extracted the texts excluding these three types of texts (NL$_T$).

IV. Answering RQ1. Which similarity model can localize buggy files the most effectively?

A. Method

To answer RQ1, we compared IRBL performance using the three document similarity models mentioned in Section II-B2. We implemented the VSM and BM25 using sklearn[2] and rank_bm25[3], respectively.

B. Result

Table III presents the experimental results for three models.

TABLE III: Comparison of similarity models

Model	Top-1	Top-5	Top-10	MRR	MAP	Retrieval time (sec)
VSM	0.312	0.615	0.735	0.452	0.371	0.8
rVSM	0.342*	0.647*	0.750*	0.484*	0.400*	0.8
BM25	**0.411***	**0.794***	**0.798***	**0.542***	**0.452***	1.3

Bold indicates the highest performance.

VSM can localize at least one buggy file in the top 10 for 73.5% of bug reports, with the MRR and MAP being 0.452 and 0.371, respectively. Meanwhile, rVSM exhibited improvements of 7.1% and 7.8%[4] in terms of MRR and MAP, respectively, compared with VSM. In other words, similar to the findings identified by BugLocator, we found that longer source files in Python projects are more likely to contain bugs. Finally, BM25 outperformed the other methods. In particular, the MRR and MAP of BM25 were 12.0% and 13.0% higher than those of rVSM, respectively. Moreover, BM25 can localize at least one buggy file in the top 10 for approximately 80% of bug reports.

We considered the time cost of each IR model. The time cost of retrieval was 0.8 seconds for VSM and rVSM but about 1.3 seconds for BM25. Kochhar et al. revealed that more than 90% of developers are satisfied if they can find a buggy entity in less than one minute [12]. Even though VSM and rVSM were faster than BM25, BM25 can still localize in about 1 second with the highest localization accuracy. Therefore, the answer to RQ1 is that BM25 is the most effective similarity model for localizing buggy files.

[2]https://scikit-learn.org/

[3]https://pypi.org/project/rank-bm25/

[4]Subsequently, the improvement rate is computed by $\frac{New-Previous}{Previous}$

V. Answering RQ2. Are all texts in bug reports effective as queries?

A. Method

To answer RQ2, we evaluated three types of queries using BM25, which proved to be the superior similarity model. The texts were separated into 18 sub-texts using the method described in Section III-C. We then compared the performance of the three types of queries to identify noisy texts in queries for IRBL. The first query was the baseline in which all texts were used as queries. Meanwhile, the second query was generated by selecting each text individually (selection). Finally, the third query was generated by removing each text from the baseline query individually (reduction). Since the number of bug reports differs for each textual feature, the performance of the baseline is different.

B. Result

Table IV presents the experimental results for different textual features.

TABLE IV: Query Effectiveness

Type	Feature	#BR	Baseline		Selection		Reduction	
			MRR	MAP	MRR	MAP	MRR	MAP
BASE	Sum	2,365	**0.542**	**0.452**	0.455	0.378	0.503	0.418
	Desc	2,365	**0.542**	**0.452**	0.503	0.418	0.455	0.378
DATA	Trace	583	**0.521**	**0.433**	0.444	0.364	0.486	0.412
	ErrMsg	1,056	**0.539**	**0.449**	0.153	0.126	0.536	0.447
	CE	2,359	**0.542**	**0.452**	0.501	0.419	0.410	0.331
	NL$_D$	2,365	**0.542**	**0.452**	0.386	0.312	0.496	0.414
INFO	Bold	37	**0.452**	**0.383**	0.379	0.310	0.400	0.337
	Guide	61	**0.541**	**0.467**	0.288	0.212	0.521	0.456
	Solution	76	**0.494**	**0.401**	0.384	0.321	0.471	0.379
	Extra	186	**0.503**	**0.414**	0.249	0.208	0.483	0.401
	EB	225	**0.489**	**0.392**	0.316	0.245	0.481	0.383
	OB	265	**0.433**	**0.344**	0.311	0.235	0.431	0.351
	ENV	441	0.468	0.375	0.094	0.077	**0.506***	**0.406***
	S2R	562	**0.463**	**0.379**	0.369	0.296	0.401	0.329
	Desc	744	**0.488**	**0.400**	0.444	0.362	0.402	0.326
TAG	Header	835	0.492	0.403	0.070	0.054	**0.498**	**0.409***
	Link	947	0.517	**0.428**	0.144	0.116	**0.518**	0.427
	Code	1,174	**0.520**	**0.437**	0.450	0.378	0.460	0.385
	NL$_T$	2,365	**0.542**	**0.452**	0.511	0.424	0.273	0.228

Bold indicates the highest performance overall. Gray highlights indicate the highest performance between selection and reduction.

BASE type. When the summary and description are individually used, the performance was not significantly better than using both (baseline). Therefore, both the summary and description are effective for queries. Some studies have indicated that the summary is vital information for generating a query [42]. However, the description is more valuable than the summary for queries because the performance of both summary reduction and description selection is higher than that of both summary selection and description reduction.

DATA type. Trace can provide important information to developers, but when only a trace is selected as a query, the MRR and MAP deteriorate by 14.8% and 16.0%, respectively. Rahman et al. also reported that the stack trace is noisy for IRBL [23]. For ErrMsg, the result is similar to the result of Trace. In the case of CE, texts can provide valuable information, but using them alone deteriorates performance. When removing the CE from the query, performance decreases, but the performance of reduction is lower than that of selection. This means that using CE is better than removing it for generating an effective query. For NL$_D$, the performances of both types of queries deteriorated. Because the reduction performance is better than the selection performance, we can infer that the NL$_D$ noise is greater in the opposite case. However, the performance of reduction is worse than that of the baseline, so removing all NL$_D$ is not suitable for queries. Rahman et al. and Mills et al. empirically demonstrated that partial NL texts are helpful for generating effective queries [43], [44]. Our results support these findings.

INFO type. Save for Desc, all selection performances were worse than those of the baselines. For Desc, the selection performance is higher than that of the reduction. Similar to the CE case, this result indicates that the texts in the Desc section can provide valuable information for effective queries.

In the case of reduction, excluding ENV, the performance deteriorated. Conversely, the performance significantly increased when the ENV texts were removed. This result suggests that queries should not contain ENV. In this incubator-mxnet#13141[5], the ENV section of the bug report contains the versions for Python, pip, and mxnet, OS, platform, hardware, and network test result. Although this information is beneficial for reproducing the bugs for developers, it is less likely to include the texts in the buggy entity.

Two additional interesting results are observed. The first is that the solution is not a vital text for an effective query, as the performance associated with reducing the solution is higher than that of selecting it. If the solution's texts contain exact buggy-file-related information, then the IRBL performance of selection could be better than that of reduction; however, our results indicate the opposite effect. For the bug report in which it was good to remove the solution, the solution fields contained human-readable bug-fixing solutions. The readable solutions could be beneficial for developers but had rare keywords for IRBL.

Second, our experimental results indicated that OB and EB are not vital for effective queries, even though previous studies indicated that OB and EB selected by humans could be used for good queries [24], [45]. These two results suggest that when generating a query for IRBL in DLSW, selecting useful words or removing unnecessary noise words from the solution, OB, and EB texts, is essential.

TAG type. Similar to other types, the performance deteriorated significantly when only one textual feature was selected. Although we expected queries selecting only CODE texts to be better than the baseline, the performance actually worsened. This result indicates that a query containing all texts is better than a query containing only CODE texts. For reduction, the MRR and MAP increased in the case of Header. The MRR differences were not significant, but those in MAP were significant. Therefore, removing heading texts can reduce developer effort when localizing all buggy files.

Summary. In our experimental result, we could confirm that no feature can provide good information alone. For all features, the performance deteriorated when used alone. Furthermore, we identified the noisy features. Environmental information and heading texts improved the performance when excluded. Therefore, we answer RQ2 by concluding that integrating all texts excluding these two types of text is effective for generating IRBL queries for DLSW.

VI. Answering RQ3. Which weights for bug-related features are optimal for achieving effective IRBL?

A. Method

To answer RQ3, we evaluated buggy file localization performance and analyzed the results by integrating the bug-related features mentioned in Section II-B3. BM25 was used for file similarity (FileSim) based on the analysis presented in Section IV. BugSim and HunkSim were also computed based on BM25 similarity. To integrate these features, we defined the following equation.

$$\begin{aligned} score(br, sf) &= f \cdot FileSim(br, sf) + b \cdot BugSim(br, sf) \\ &\quad + c \cdot Comm(sf, k) + s \cdot Strace(br, sf) \\ &\quad + h \cdot HunkSim(br, sf) \end{aligned} \tag{8}$$

The constraint is that the sum of weights such as b, c, s, h, and f is one. The k range is $\{15, 30, 60, 90\}$. We normalized each score for each feature using $\frac{max-value}{max-min}$. The final score of each source file was computed by changing each weight in increments of 0.1 and the best-performing weight was selected.

We compare between the baseline performance using only file similarity (Base) and the best performance with the optimal weight for each bug-related feature (Best).

B. Result

Table V presents the experimental results regarding the best weights in terms of performance. For the Strace score, it is necessary to assess the impact of a stack trace in a bug report. We classified the bug reports into stack trace (BR-ST) and other (BR-NST) and then evaluated the performance for each class. There were 583 BR-ST and 1,782 BR-NST. Since the comparison dataset was unbalanced, we randomly selected 583 BR-NST (BR-NST$_R$).

BR-NST. The best performance was obtained when weights of 0.0, 0.4, 0.3, and 0.3, were assigned to the scores for BugSim, Comm, HunkSim, and FilsSim, respectively, with a k value of 30 days. When using the optimal weights, MRR and MAP were significantly improved by 11% and 14%,

[5]https://github.com/apache/incubator-mxnet/issues/13141

TABLE V: Performance of buggy file localization with the optimal weight

BR Type (#BR)	Target	Best Weight b	c (k)	s	h	f	MRR	MAP
BR-NST (1,782)	BASE	0	0 (-)	-	0	1.0	0.549	0.458
	BEST	0	0.4 (30)	-	0.3	0.3	**0.609***	**0.523***
BR-NST$_R$ (583)	BASE	0	0 (-)	-	0	1.0	0.552	0.463
	BEST	0	0.4 (30)	-	0.3	0.3	**0.613***	**0.523***
BR-ST (583)	BASE	0	0 (-)	0	0	1.0	0.521	0.433
	w/o S	0	0.6 (15)	0	0.1	0.3	0.555*	0.471*
	BEST	0	0.4 (15)	0.2	0.1	0.3	**0.632***	**0.525***
All (2,365)	BASE	0	0 (-)	0	0	1.0	0.542	0.452
	BEST	0	0.2 (60)	0.4	0.2	0.2	**0.613***	**0.520***

Bold indicates the highest performance. Gray highlights indicate the highest weight.

respectively. Setting k to 30 days indicates that a source file modified within 30 days is likely to be buggy on average. The optimal weight of Comm was the highest, but the optimal weight of BugSim was zero. Similar results were obtained for the random sample, BR-NST$_R$. Our experimental results indicate that bug report similarity is not practical for localizing buggy files as buggy files are likely to differ, even if the bug reports are similar. In addition, the influence of Comm was small (0.0 or 0.2) in Youm et al. [46], but was confirmed to be large in DLSW.

BR-ST. The weights for each feature are set to 0, 0.4, 0.2, 0.1, and 0.3 for the best IRBL performance. These settings yield MRR and MAP improvements of 21%. A similar trend occurred with BR-NST when the weight of BugSim was set to zero. Without the Strace score (w/o S), the performance was lower than when using the Strace score. In addition, the best performance demonstrated by BR-ST was much better than BR-NST$_R$ and BR-NST. Therefore, the Strace score should be used to localize buggy files when a bug report includes a stack trace.

ALL. We can answer RQ3; for the best possible IRBL performance for overall bug reports, the weights for each feature should be set as 0, 0.2, 0.4, 0.2, and 0.2 with k set to 60 days. The MRR and MAP when using all features excluding BugSim with these weights are higher than those when using only FileSim by 13% and 15%, respectively. Regarding the MRR metric, we can infer that IRBL with optimal weights can localize at least one buggy file at rank 1.63 (1/0.613).

VII. Answering RQ4. How much performance can be expected from IRBL for buggy function localization?

A. Method

To answer RQ4, we evaluated function localization performance using three methods we designed. We normalized each score for each feature using $\frac{max-value}{max-min}$. The three methods for comparison are summarized below.

1) Func: Ranking functions by the similarity between functions and bug reports.
2) SF+Func: On the similarity score of the function to the bug report (Func), the similarity score between the bug report and the source file is added.
3) BL+Func: IRBL-based source file scores are computed based on the optimal weights validated in Section VI. The final scores are sums of the IRBL scores and the similarity between functions and bug reports.

B. Result

Table VI presents the function localization performances of the three methods.

TABLE VI: Performance of buggy function localization

Type	Top-1	Top-5	Top-10	MRR	MAP
Func	0.220	0.401	0.494	0.311	0.237
SF+Func	0.233*	0.435*	0.521*	0.331*	0.262*
BL+Func	**0.247***	**0.458***	**0.550***	**0.348***	**0.273***

Bold indicates the highest performance.

A total of 1,817 bug reports were resolved by fixing functions. The BL+Func method exhibited the best performance. The MRR and MAP for BL+Func were 11.9% and 15.2% higher than those for Func, respectively. By using only function similarity for IRBL, buggy functions can be localized in the top 10 for about 49% of bug reports. Integrating file and function similarity scores can localize buggy functions in the top 10 for 52% of bug reports. Because the IRBL with optimal weights yields the best performance for buggy file localization, integrating IRBL scores with function similarity is desirable.

We can answer RQ4 by stating that the IRBL can localize buggy function in the top 10 for 56% of bug reports when integrating file scores with function similarity. Considering the MRR metric, we can infer that the IRBL with optimal weights can localize at least one buggy file into a 2.87 rank (1/0.348).

Previous paper has reported that the MRR and MAP values for buggy function (method in Java) localization performance are lower than 0.3 and 0.2, respectively [29]. However, in DLSW written in Python, the function localization performance was higher than the reported performance for Java. Therefore, DLSW developers can trust IRBL for function localization more than Java project developers.

VIII. Discussion

A. Are there differences between DL bugs and others bugs?

Our study indicated that IRBL can be effectively used for bug localization in DLSW as the experimental results showed similar or higher IRBL performance than other studies [28], [29], [46]. This suggests that developers can effectively use IRBL in both non-DLSW and DLSW.

The bug reports in DLSW can be divided into DL-related bugs (e.g., model or data shape) and non-DL-related bugs (e.g., logging or tests). If a difference exists between the two types of bugs, additional research might be required to obtain a suitable strategy for each bug type. We performed a preliminary study to evaluate the performance for each type of bug. We selected 100 DL bug reports and 100 non-DL bug reports through random sampling and manual investigation. Following Humbatova et al [16]. and Jia et al. [47], we classified DL-related bugs (DL bugs) as those that affect DL functionality and non-DL-related bugs (NDL bugs) as those that do not directly affect DL functionality. After labeling, we evaluated the performance by considering the four RQs. Table VII presents the results of this study.

TABLE VII: Comparison between DL and NDL bugs

(a) Comparison of similarity models

Model	DL bugs (100)				NDL bugs (100)			
	Top-1	Top-5	MRR	MAP	Top-1	Top-5	MRR	MAP
VSM	0.180	0.450	0.312	0.229	0.260	0.540	0.387	0.323
rVSM	0.190	0.490*	0.336*	0.254*	0.280*	0.580*	0.422*	0.357*
BM25	**0.280***	**0.600***	**0.423***	**0.336***	**0.410***	**0.650***	**0.523***	**0.447***

(b) Query effectiveness (SEL: Selection, RED: Reduction)

Type	Entity	#BR (Ratio)		MAP for DL bugs			MAP for NDL bugs		
		DL bugs	NDL bugs	BASE	SEL	RED	BASE	SEL	RED
DATA	Trace	**30 (30%)**	27 (27%)	**0.269**	0.256	0.266	**0.507**	0.387	0.413
	ErrMsg	**61 (61%)**	43 (43%)	**0.307**	0.082	**0.307**	**0.510**	0.091	0.495
	CE	100 (100%)	100 (100%)	**0.336**	0.329	0.221	**0.447**	0.390	0.267
	NL_D	100 (100%)	100 (100%)	0.336	0.207	0.314	**0.447**	0.254	0.398
INFO	Bold	**4 (4%)**	3 (3%)	0.304	0.298	**0.348***	0.515	**0.572***	0.180
	Guide	4 (4%)	**14 (14%)**	**0.289**	**0.289**	0.244	**0.618**	0.184	0.564
	Solution	**10 (10%)**	8 (8%)	**0.543**	0.406	**0.543**	0.632	0.425	**0.634**
	Extra	16 (16%)	16 (16%)	0.174	0.079	**0.179**	**0.669**	0.395	0.636
	EB	**16 (16%)**	7 (7%)	**0.280**	0.224	0.236	0.644	0.226	**0.678***
	OB	**23 (23%)**	20 (20%)	**0.363**	0.345	0.296	**0.480**	0.194	0.454
	ENV	**46 (46%)**	32 (32%)	0.327	0.051	**0.387***	0.504	0.072	**0.518***
	S2R	68 (68%)	**70 (70%)**	**0.269**	0.182	0.252	**0.424**	0.279	0.363
	Desc	90 (90%)	**92 (92%)**	**0.321**	0.303	0.268	**0.446**	0.343	0.328
TAG	Link	40 (40%)	**55 (55%)**	**0.243**	0.099	0.237	**0.406**	0.150	0.387
	Code	**62 (62%)**	61 (61%)	**0.354**	0.308	0.315	**0.449**	0.383	0.401
	Header	100 (100%)	100 (100%)	0.336	0.031	**0.339**	0.447	0.040	**0.449**
	NL_T	100 (100%)	100 (100%)	**0.336**	0.318	0.222	**0.447**	0.411	0.284

(c) Performance of buggy file localization with the optimal weight

BR	Bug (#BR)	Type	Optimal Weight					MRR	MAP
			b	c (k)	s	h	f		
ALL	DL	BASE	0	0	0	0	1.0	0.423	0.336
		BEST	0	0.3 (30)	0.4	0.1	0.2	**0.571***	**0.452***
	NDL	BASE	0	0	0	0	1.0	0.523	0.447
		BEST	0	0.3 (30)	0.1	0.5	0.1	**0.634***	**0.573***

(d) Performance of buggy function localization

Method	DL (80)			DL (67)			NDL (67)		
	Top-10	MRR	MAP	Top-10	MRR	MAP	Top-10	MRR	MAP
Func	0.400	0.215	0.155	0.403	0.229	0.158	0.448	0.337	0.286
SF+Func	0.362	0.219	0.163	0.373	0.234	0.166	0.493*	0.334	0.302*
BL+Func	**0.400**	**0.247***	**0.187***	**0.403**	**0.253***	**0.182***	**0.537***	**0.360***	**0.321***

Differences between DL and NDL bugs. Table VIIa presents the experimental results for the three document similarity models. BM25 outperforms the other models for both types. Additionally, the buggy file localization performance of NDL bugs is higher than that of DL bugs. Table VIIb presents the impact of textual features based on MAP. There are different distributions of textual features. DL bugs have more bug reports with Trace, ErrMsg, Bold, Solution, EB, OB, ENV, and Code than NDL cases. NDL cases have more bug reports with Guide, S2R, Desc, and Link. The informative texts, such as OB, EB, Trace, and Solution, required to fix bugs were ranked higher in the DL bug reports. A comparison of the query effectiveness between DL bugs and NDL bugs confirmed that ENV acted as noise for both types of bugs, similar to the results analyzed for DLSW in Section V. However, two points differed from the results discussed in Section V. For DL bugs, removing Bold or Solution yields significantly higher performance than the baseline and selection cases. For NDL bugs, removing EB is better than the baseline case and EB selection, and selecting Bold is better than the two cases. It is difficult to draw definitive conclusions because the sample does not represent all DL and NDL bugs. However, this result implies that there would be differences between the impactful texts for each type of bug for IRBL. Table VIIc presents the IRBL performance for buggy file localization with the optimal weights. Similar to the DLSW comprehensive bug reports, similar bug report scores did not have a significant impact. However, there is a slight difference in weights for the Strace and HunkSim scores. As shown in Table VIId, when localizing buggy functions, it is desirable to combine the function similarity and IRBL results.

In short, there are different characteristics between DL and NDL bugs in the perspective of IRBL and the IRBL performance for DL bugs is lower than that for NDL bugs even though the DL bug reports contain more informative texts.

IRBL for DL bugs Our preliminary results reveal two important considerations for IRBL for DL bugs. The first is that similar bug reports are not helpful in localizing DL bugs. We expect a diversely written bug report even it shares the same defect files. We must find more valuable information from the bug reports in the past, upon the mere similarity. The second is that IRBL performance is still insufficient despite the DL bug report having more information for humans to see. The results of 200 samples revealed that DL bugs have better information than NDL bugs. We should review this information more closely and improve the query quality [48].

B. Suggestions for DLSW developers for using IRBL

Based on our experimental results, we suggest several strategies for effectively utilizing IRBL in DLSW.

1) For FileSim, BM25 is better than VSM and rVSM.
2) When generating a query, environmental information and heading should be removed from the initial query.
3) If possible, we recommend setting optimal weights by analyzing historical project datasets. If there is no sufficient dataset, then setting weights for Comm, Strace, HunkSim, and FileSim of 0.2, 0.4, 0.2, 0.2, respectively, and setting k for Comm to 60 is a feasible solution.
4) IRBL can also effectively localize buggy functions for Python DLSW.

C. Suggestions for IRBL researchers focusing on DLSW

Based on the results of this study, we propose the following three suggestions for IRBL researchers for DLSW.

1) We must automatically classify bug reports into DL and NDL types because the impacts of textual features are different between these bug types.
2) We must improve IRBL performance for DL bugs because the performance for DL bugs is lower than that for NDL bugs.

3) Our study for buggy function localization for DLSW was relatively good compared to non-DLSW. However, there is room for improvement compared to buggy file localization. Thus, follow-up studies to improve the performance of buggy function localization are required.

D. Threats to validity

There are some internal validity issues related to experimental errors. Because existing IRBL tools were developed for Java projects, we implemented each bug-related feature using popular open-source packages. Because we adopted a version-based search space, there are bugs without ground-truth files. We removed these bug reports because we could not evaluate their performance. We extracted textual features from bug reports with heuristically defined regular expressions and headings, so the information may not have been extracted with 100% precision. However, it is a reasonable heuristic from an automation perspective because it would introduce another type of overhead for developers to select these features.

There are some external validity issues related to the generalizability of our results. Although the experimental results might vary by project, our results should be similar for other projects and bug reports, as we presented average performance values based on approximately 2,000 bug reports and performed statistical analysis. Because DLSW is primarily developed in Python, we experimented on only Python bugs. We plan to extend our study to multiple languages in the future. The results of RQ3 are difficult to fully generalize since different bug reports might require different optimal weights. Accordingly, a more detailed analysis of RQ3 will be performed using learning to rank [28]. The final threat involves the discussion. The results cannot be fully generalized, as we ran the analysis with 200 samples. In the future, after classifying all the bugs using Denchmark [10], we will present more detailed differences between the DL bugs and NDL bugs.

IX. Related works

A. DLSW debugging

Some studies have revealed commonalities and differences in types, symptoms, and root causes of bugs in DLSW compared to traditional software. Zhang et al. investigated the symptoms and causes of bugs that occur when using Tensorflow [3]. Islam et al. demonstrated that DL frameworks with similar bug distributions share a common anti-pattern [4]. Islam et al. classified bug fix patterns, analyzed the distribution of fix patterns by bug type, and revealed that there are fix patterns that differ significantly from those in traditional software [7]. Humbatova et al. defined words related to DL, and analyzed and categorized selected DL-specific bugs [16].

Along with studies on the characteristics of DLSW bugs, DLSW debugging techniques are also being actively studied. Odena et al. proposed TensorFuzz to discover bugs in neural networks that occur only for rare inputs [49]. Humbatova et al. proposed 24 DL mutation operators for a DL system based on DL fault taxonomy [50]. Because operating DLSW is expensive, studies on finding bugs in code through static analysis have also been performed. Dolby et al. proposed Adriane, which detects errors by tracking and analyzing a tensor based on WALA [51]. Zhang et al. proposed the DEBAR program, which can detect numerical bugs in neural networks through static analysis using two abstraction techniques [52]. Schoop et al. proposed Umlaut, which can detect errors by comparing the program structures and model behaviors of DLSW [53]. Previous studies have focused on bugs in neural networks. Our analysis covered all scopes written in source files and functions (both neural networks and the functionality within DLSW) to perform IRBL from the perspective of DLSW.

B. IRBL validation

IRBL has been validated in various environments with consideration for similarity models and search space settings. Polissety et al. examined the performance of various IRBL models with different similarity models based on machine learning techniques [54] because recent IRBL studies used deep learning-based document similarity [55], [56]. Saha et al. demonstrated that structural information provides less benefit for C software than Java software in terms of bug localization [57]. Akbar et al. evaluated three generations of bug localization models on BugzBook, which is a large-scale dataset consisting of Java, C/C++, and Python source code files [58]. Lee et al. [40] and Kim et al. [59] pointed out the problem of datasets with test files for IRBL evaluation. We evaluated IRBL performance with a focus on DLSW using three similarity models: VSM, rVSM, and BM25.

Some studies have analyzed and verified IRBL model performance for different query types because search results can vary significantly depending on query types. The experimental results presented by Mills et al. demonstrate the plausibility of query reformulation using text in a bug report [44]. Chaparro et al. showed that the high-quality query for IRBL contains title, the observed behavior, expected behavior, steps to reproduce, and code snippets [24]. We also validated IRBL performance from the perspective of the query texts.

X. Conclusion

Recent IRBL studies have validated the IRBL on general software. Since the need for debugging DLSW is increasing with its popularity, investigation of IRBL performance in DLSW is requisite. Our study evaluated IRBL performance from four perspectives: document similarity, query generation, bug-related features, and function localization. Our experimental results yielded recommendations for developers who wish to use IRBL in DLSW and research directions for researchers who want to study IRBL in DLSW. We validated the application of IRBL on DLSW with various aspects.

We share all reproducible codes and dataset on `https://github.com/RosePasta/IRBL_for_DLSW`.

Acknowledgment

This work was supported by the National Research Foundation of Korea (NRF) grant funded by the Korea government (MSIT)(2019R1A2C2006411, 2021R1A6A3A01086997)

REFERENCES

[1] Dario Amodei, Chris Olah, Jacob Steinhardt, Paul Christiano, John Schulman, and Dan Mané. Concrete problems in ai safety. *arXiv preprint arXiv:1606.06565*, 2016.

[2] Saleema Amershi, Andrew Begel, Christian Bird, Robert DeLine, Harald Gall, Ece Kamar, Nachiappan Nagappan, Besmira Nushi, and Thomas Zimmermann. Software engineering for machine learning: A case study. In *Proceedings of the International Conference on Software Engineering: Software Engineering in Practice (ICSE-SEIP)*, pages 291–300. IEEE, 2019.

[3] Yuhao Zhang, Yifan Chen, Shing-Chi Cheung, Yingfei Xiong, and Lu Zhang. An empirical study on tensorflow program bugs. In *Proceedings of the International Symposium on Software Testing and Analysis (ISSTA)*, pages 129–140, 2018.

[4] Md Johirul Islam, Giang Nguyen, Rangeet Pan, and Hridesh Rajan. A comprehensive study on deep learning bug characteristics. In *Proceedings of the ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE)*, pages 510–520, 2019.

[5] Giang Nguyen, Stefan Dlugolinsky, Martin Bobák, Viet Tran, Álvaro López García, Ignacio Heredia, Peter Malík, and Ladislav Hluchỳ. Machine learning and deep learning frameworks and libraries for large-scale data mining: a survey. *Artificial Intelligence Review*, 52(1):77–124, 2019.

[6] Hadhemi Jebnoun, Houssem Ben Braiek, Mohammad Masudur Rahman, and Foutse Khomh. The scent of deep learning code: An empirical study. In *Proceedings of the International Conference on Mining Software Repositories (MSR)*, pages 420–430, 2020.

[7] Md Johirul Islam, Rangeet Pan, Giang Nguyen, and Hridesh Rajan. Repairing deep neural networks: Fix patterns and challenges. *arXiv preprint arXiv:2005.00972*, 2020.

[8] Mohammad Wardat, Wei Le, and Hridesh Rajan. Deeplocalize: Fault localization for deep neural networks. In *Proceedings of the International Conference on Software Engineering (ICSE)*, pages 251–262. IEEE, 2021.

[9] Thomas Zimmermann, Rahul Premraj, Nicolas Bettenburg, Sascha Just, Adrian Schroter, and Cathrin Weiss. What makes a good bug report? *Transactions on Software Engineering (TSE)*, 36(5):618–643, 2010.

[10] Misoo Kim, Youngkyoung Kim, and Eunseok Lee. Denchmark: A bug benchmark of deep learning-related software. In *Proceedings of the International Conference on Mining Software Repositories (MSR)*, pages 540–544. IEEE, 2021.

[11] Jian Zhou, Hongyu Zhang, and David Lo. Where should the bugs be fixed? more accurate information retrieval-based bug localization based on bug reports. In *Proceedings of the International Conference on Software Engineering (ICSE)*, pages 14–24. IEEE, 2012.

[12] Pavneet Singh Kochhar, Xin Xia, David Lo, and Shanping Li. Practitioners' expectations on automated fault localization. In *Proceedings of the International Symposium on Software Testing and Analysis (ISSTA)*, pages 165–176. ACM, 2016.

[13] W Eric Wong, Ruizhi Gao, Yihao Li, Rui Abreu, and Franz Wotawa. A survey on software fault localization. *Transactions on Software Engineering (TSE)*, 42(8):707–740, 2016.

[14] Juan Manuel Florez, Oscar Chaparro, Christoph Treude, and Andrian Marcus. Combining query reduction and expansion for text-retrieval-based bug localization. In *Proceedings of the International Conference on Software Analysis, Evolution and Reengineering (SANER)*, pages 166–176. IEEE, 2021.

[15] Junxiao Han, Emad Shihab, Zhiyuan Wan, Shuiguang Deng, and Xin Xia. What do programmers discuss about deep learning frameworks. *Empirical Software Engineering (ESE)*, 25(4):2694–2747, 2020.

[16] Nargiz Humbatova, Gunel Jahangirova, Gabriele Bavota, Vincenzo Riccio, Andrea Stocco, and Paolo Tonella. Taxonomy of real faults in deep learning systems. In *Proceedings of the International Conference on Software Engineering (ICSE)*, pages 1110–1121, 2020.

[17] David Sculley, Gary Holt, Daniel Golovin, Eugene Davydov, Todd Phillips, Dietmar Ebner, Vinay Chaudhary, Michael Young, Jean-Francois Crespo, and Dan Dennison. Hidden technical debt in machine learning systems. *Advances in Neural Information Processing Systems*, 28:2503–2511, 2015.

[18] Mozhan Soltani, Felienne Hermans, and Thomas Bäck. The significance of bug report elements. *Empirical Software Engineering (ESE)*, 25(6):5255–5294, 2020.

[19] Zejun Zhang, Yanming Yang, Xin Xia, David Lo, Xiaoxue Ren, and John Grundy. Unveiling the mystery of api evolution in deep learning frameworks a case study of tensorflow 2. In *Proceedings of the International Conference on Software Engineering: Software Engineering in Practice (ICSE-SEIP)*, pages 238–247. IEEE, 2021.

[20] Xufan Zhang, Yilin Yang, Yang Feng, and Zhenyu Chen. Software engineering practice in the development of deep learning applications. *arXiv preprint arXiv:1910.03156*, 2019.

[21] Chakkrit Tantithamthavorn, Surafel Lemma Abebe, Ahmed E Hassan, Akinori Ihara, and Kenichi Matsumoto. The impact of ir-based classifier configuration on the performance and the effort of method-level bug localization. *Information and Software Technology (IST)*, 102:160–174, 2018.

[22] Md Masudur Rahman, Saikat Chakraborty, and Baishakhi Ray. Which similarity metric to use for software documents? a study on information retrieval based software engineering tasks. In *Proceedings of the International Conference on Software Engineering: Companion Proceeedings (ICSE-C)*, pages 335–336, 2018.

[23] Mohammad Masudur Rahman and Chanchal K Roy. Improving ir-based bug localization with context-aware query reformulation. In *Proceedings of the ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE)*, pages 621–632, 2018.

[24] Oscar Chaparro, Juan Manuel Florez, and Andrian Marcus. Using bug descriptions to reformulate queries during text-retrieval-based bug localization. *Empirical Software Engineering (ESE)*, 24(5):2947–3007, 2019.

[25] Ripon K Saha, Matthew Lease, Sarfraz Khurshid, and Dewayne E Perry. Improving bug localization using structured information retrieval. In *Proceedings of the International Conference on Automated Software Engineering (ASE)*, pages 345–355. IEEE, 2013.

[26] Chu-Pan Wong, Yingfei Xiong, Hongyu Zhang, Dan Hao, Lu Zhang, and Hong Mei. Boosting bug-report-oriented fault localization with segmentation and stack-trace analysis. In *Proceedings of the International Conference on Software Maintenance and Evolution (ICSME)*, pages 181–190. IEEE, 2014.

[27] Bunyamin Sisman and Avinash C Kak. Incorporating version histories in information retrieval based bug localization. In *Proceedings of the Working Conference on Mining Software Repositories (MSR)*, pages 50–59. IEEE, 2012.

[28] Xin Ye, Razvan Bunescu, and Chang Liu. Mapping bug reports to relevant files: A ranking model, a fine-grained benchmark, and feature evaluation. *Transactions on Software Engineering (TSE)*, 42(4):379–402, 2016.

[29] Wen Zhang, Ziqiang Li, Qing Wang, and Juan Li. Finelocator: A novel approach to method-level fine-grained bug localization by query expansion. *Information and Software Technology (IST)*, 110:121–135, 2019.

[30] Rafi Almhana, Marouane Kessentini, and Wiem Mkaouer. Method-level bug localization using hybrid multi-objective search. *Information and Software Technology (IST)*, 131:106474, 2021.

[31] Misoo Kim and Eunseok Lee. Are datasets for information retrieval-based bug localization techniques trustworthy? *Empirical Software Engineering (ESE)*, 26(3):1–66, 2021.

[32] Gede Artha Azriadi Prana, Christoph Treude, Ferdian Thung, Thushari Atapattu, and David Lo. Categorizing the content of github readme files. *Empirical Software Engineering (ESE)*, 24(3):1296–1327, 2019.

[33] Luca Ponzanelli, Andrea Mocci, and Michele Lanza. Stormed: Stack overflow ready made data. In *Proceedings of the IEEE/ACM Working Conference on Mining Software Repositories (MSR)*, pages 474–477. IEEE, 2015.

[34] Md Ahasanuzzaman, Muhammad Asaduzzaman, Chanchal K Roy, and Kevin A Schneider. Classifying stack overflow posts on api issues. In *Proceedings of the International Conference on Software Analysis, Evolution and Reengineering (SANER)*, pages 244–254. IEEE, 2018.

[35] Di Wu, Xiao-Yuan Jing, Hongyu Zhang, Bing Li, Yu Xie, and Baowen Xu. Generating api tags for tutorial fragments from stack overflow. *Empirical Software Engineering (ESE)*, 26(4):1–37, 2021.

[36] Stephen W Thomas, Meiyappan Nagappan, Dorothea Blostein, and Ahmed E Hassan. The impact of classifier configuration and classifier combination on bug localization. *Transactions on Software Engineering (TSE)*, 39(10):1427–1443, 2013.

[37] Klaus Changsun Youm, June Ahn, Jeongho Kim, and Eunseok Lee. Bug localization based on code change histories and bug reports.

In *Proceedings of the Asia-Pacific Software Engineering Conference (APSEC)*, pages 190–197. IEEE, 2015.

[38] Ming Wen, Rongxin Wu, and Shing-Chi Cheung. Locus: Locating bugs from software changes. In *Proceedings of the International Conference on Automated Software Engineering (ASE)*, pages 262–273. IEEE, 2016.

[39] Gregory W Corder and Dale I Foreman. *Nonparametric statistics: A step-by-step approach*. John Wiley & Sons, 2014.

[40] Jaekwon Lee, Dongsun Kim, Tegawendé F Bissyandé, Woosung Jung, and Yves Le Traon. Bench4bl: reproducibility study on the performance of ir-based bug localization. In *Proceedings of the International Symposium on Software Testing and Analysis (ISSTA)*, pages 61–72, 2018.

[41] Nikhil Ketkar and Eder Santana. *Deep learning with Python*, volume 1. Springer, 2017.

[42] Mohammad Masudur Rahman and Chanchal K Roy. Improved query reformulation for concept location using coderank and document structures. In *Proceedings of the International Conference on Automated Software Engineering (ASE)*, pages 428–439. IEEE, 2017.

[43] Mohammad Masudur Rahman, Foutse Khomh, Shamima Yeasmin, and Chanchal K Roy. The forgotten role of search queries in ir-based bug localization: an empirical study. *Empirical Software Engineering (ESE)*, 26(6):1–56, 2021.

[44] Chris Mills, Jevgenija Pantiuchina, Esteban Parra, Gabriele Bavota, and Sonia Haiduc. Are bug reports enough for text retrieval-based bug localization? In *Proceedings of the International Conference on Software Maintenance and Evolution (ICSME)*, pages 381–392. IEEE, 2018.

[45] Misoo Kim and Eunseok Lee. Manq: Many-objective optimization-based automatic query reduction for ir-based bug localization. *Information and Software Technology (IST)*, 125:106334, 2020.

[46] Klaus Changsun Youm, June Ahn, and Eunseok Lee. Improved bug localization based on code change histories and bug reports. *Information and Software Technology (IST)*, 82:177–192, 2017.

[47] Li Jia, Hao Zhong, Xiaoyin Wang, Linpeng Huang, and Xuansheng Lu. The symptoms, causes, and repairs of bugs inside a deep learning library. *Journal of Systems and Software (JSS)*, 177:110935, 2021.

[48] Misoo Kim and Eunseok Lee. A novel approach to automatic query reformulation for ir-based bug localization. In *Proceedings of the ACM/SIGAPP Symposium on Applied Computing (SAC)*, pages 1752–1759, 2019.

[49] Augustus Odena, Catherine Olsson, David Andersen, and Ian Goodfellow. Tensorfuzz: Debugging neural networks with coverage-guided fuzzing. In *Proceedings of the International Conference on Machine Learning (ICML)*, pages 4901–4911. PMLR, 2019.

[50] Nargiz Humbatova, Gunel Jahangirova, and Paolo Tonella. Deepcrime: mutation testing of deep learning systems based on real faults. In *Proceedings of the International Symposium on Software Testing and Analysis (ISSTA)*, pages 67–78, 2021.

[51] Julian Dolby, Avraham Shinnar, Allison Allain, and Jenna Reinen. Ariadne: analysis for machine learning programs. In *Proceedings of the International Workshop on Machine Learning and Programming Languages*, pages 1–10, 2018.

[52] Yuhao Zhang, Luyao Ren, Liqian Chen, Yingfei Xiong, Shing-Chi Cheung, and Tao Xie. Detecting numerical bugs in neural network architectures. In *Proceedings of the ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE)*, pages 826–837, 2020.

[53] Eldon Schoop, Forrest Huang, and Bjoern Hartmann. Umlaut: Debugging deep learning programs using program structure and model behavior. In *Proceedings of the CHI Conference on Human Factors in Computing Systems*, pages 1–16, 2021.

[54] Sravya Polisetty, Andriy Miranskyy, and Ayşe Başar. On usefulness of the deep-learning-based bug localization models to practitioners. In *Proceedings of the International Conference on Predictive Models and Data Analytics in Software Engineering*, pages 16–25, 2019.

[55] Youngkyoung Kim, Misoo Kim, and Eunseok Lee. Feature combination to alleviate hubness problem of source code representation for bug localization. In *Proceedings of the Asia-Pacific Software Engineering Conference (APSEC)*, pages 511–512. IEEE, 2020.

[56] Jinglei Zhang, Rui Xie, Wei Ye, Yuhan Zhang, and Shikun Zhang. Exploiting code knowledge graph for bug localization via bi-directional attention. In *Proceedings of the International Conference on Program Comprehension (ICPC)*, pages 219–229, 2020.

[57] Ripon K Saha, Julia Lawall, Sarfraz Khurshid, and Dewayne E Perry. On the effectiveness of information retrieval based bug localization for c programs. In *Proceedings of the International Conference on Software Maintenance and Evolution (ICSME)*, pages 161–170. IEEE, 2014.

[58] Shayan A Akbar and Avinash C Kak. A large-scale comparative evaluation of ir-based tools for bug localization. In *Proceedings of the International Conference on Mining Software Repositories (MSR)*, pages 21–31, 2020.

[59] Misoo Kim and Eunseok Lee. Poster: Are information retrieval-based bug localization techniques trustworthy? In *Proceedings of the International Conference on Software Engineering: Companion (ICSE-C)*, pages 248–249. IEEE, 2018.