The Journal of Systems & Software 178 (2021) 110986

Contents lists available at ScienceDirect

The Journal of Systems & Software

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

An extensive study on smell-aware bug localization☆

Aoi Takahashi, Natthawute Sae-Lim, Shinpei Hayashi *, Motoshi Saeki

School of Computing, Tokyo Institute of Technology, Tokyo, 152–8550, Japan

ARTICLE INFO

Article history:
Received 24 May 2020
Received in revised form 12 April 2021
Accepted 14 April 2021
Available online 24 April 2021

Keywords:
Bug localization
Code smell
Information retrieval

ABSTRACT

Bug localization is an important aspect of software maintenance because it can locate modules that should be changed to fix a specific bug. Our previous study showed that the accuracy of the information retrieval (IR)-based bug localization technique improved when used in combination with code smell information. Although this technique showed promise, the study showed limited usefulness because of the small number of: (1) projects in the dataset, (2) types of smell information, and (3) baseline bug localization techniques used for assessment. This paper presents an extension of our previous experiments on Bench4BL, the largest bug localization benchmark dataset available for bug localization. In addition, we generalized the smell-aware bug localization technique to allow different configurations of smell information, which were combined with various bug localization techniques. Our results confirmed that our technique can improve the performance of IR-based bug localization techniques for the class level even when large datasets are processed. Furthermore, because of the optimized configuration of the smell information, our technique can enhance the performance of most state-of-the-art bug localization techniques.

1. Introduction

Bug localization is the process of identifying the locations of a given bug. Because it can be a tedious task in large-scale software development projects, many ideas have been proposed to automate this process using software development information. For instance, we can identify the locations of a bug using the description of bug reports, i.e., information retrieval (IR)-based (Lukins et al., 2010; Nguyen et al., 2011), or execution traces, i.e., dynamic analysis (Wong et al., 2014). To improve the bug localization accuracy, many hybrid techniques that combine a base technique with additional information have been proposed. For example, BugLocator (Zhou et al., 2012) combined similar bug reports that were fixed in the past with an IR-based technique. BLUiR (Saha et al., 2013) incorporated structural information in addition to using similar bug reports from the past. AmaLgam (Wang and Lo, 2014) combined the version history, report similarity, and structural information.

Although these techniques can significantly improve the bug localization accuracy, they can only be used when sufficient additional information is available. Moreover, most existing techniques do not consider the likelihood of each module containing a bug and treat all modules equally, which may lower the accuracy of bug localization. To this end, we previously proposed a smell-aware bug localization technique to improve the IR-based bug localization accuracy using code smell information (Takahashi et al., 2018). The motivation behind our approach is that modules with code smells have been found to be changed more often and fault-prone (Khomh et al., 2012; Guerrouj et al., 2017). In addition, our technique does not require additional information as code smells can be directly detected from the source code.

Although our technique can significantly enhance the bug localization performance, the previous study still has limitations that need to be addressed. First, we experimented on only four open-source systems. This small number of targeted systems means that our results may be difficult to generalize. Second, we only used one set of configurations for the technique, even though there were many possible options. Thus, our previously reported result might not be optimal. Finally, we combined our technique with only one base bug localization technique. Therefore, it remains unclear whether our technique is applicable to other bug localization techniques.

The study presented in this paper is an extension of our previous study with the objective of overcoming these limitations. First, we replicated our study on Bench4BL (Lee et al., 2018), which is the largest dataset available for bug localization. Second, we generalized the smell-aware bug localization technique and conducted studies with different configurations to find the best configurations. Finally, we combined the smell-aware bug localization technique with different base techniques provided

☆ Editor: Gabriele Bavota.

* Corresponding author.
E-mail addresses: takahashi-a-at@se.cs.titech.ac.jp (A. Takahashi), natthawute@se.cs.titech.ac.jp (N. Sae-Lim), hayashi@c.titech.ac.jp (S. Hayashi), saeki@se.cs.titech.ac.jp (M. Saeki).

https://doi.org/10.1016/j.jss.2021.110986

by Bench4BL to assess whether our technique could improve state-of-the-art bug localization techniques.

The main contributions of this paper can be summarized as follows:

- We replicate the smell-aware bug localization technique at the class level and show that it is effective even for processing a large-scale dataset.
- We generalize the smell-aware bug localization technique to allow different configurations and present the optimal configuration of the technique.
- We combine the smell-aware bug localization technique with different base bug localization techniques and show that it can improve their performance.

The remainder of this paper is organized as follows. First, we provide preliminary information about IR-based bug localization and code smells in the next section. Next, we summarize related work pertaining to empirical studies of bug localization in Section 3. We describe our smell-aware bug localization technique in Section 4. In Section 5, we provide the details of our study and present the results. Threats to validity are discussed in Section 6. Discussions of this work are presented in Section 7. Finally, we provide our conclusions in Section 8.

2. Background

2.1. IR-based bug localization and its extensions

Bug localization is the process of identifying the location of the source code that should be modified to fix a specific bug. Bug localization is challenging, especially in large-scale software systems. Therefore, automated bug localization techniques can help developers save time during such tedious processes.

IR-based bug localization techniques accept a bug report and the source code of a specific version as inputs. The approaches then determine the similarity between the bug report and source code and generate a ranking of modules based on this similarity. Developers are expected to use these rankings to help them perform bug-fixing tasks.

In IR-based bug localization, the following steps were conducted to quantify similarity.

1. **Corpus generation.** To run the approach, the text of each module needs to be processed. The test is regarded as a sequence of tokens. In addition, compound words such as *isCommitable* are divided into *is* and *Commitable*. They are then processed using the standard step as in natural language processing tasks, such as stemming or stop word removal for each module.
2. **Indexing.** The next step is to perform indexing on the generated corpus. Specifically, approaches such as the term frequency–inverse document frequency (TF–IDF) are applied. The approach calculates the importance of each word in each module. For example, in the case of using TF–IDF, the importance of each word is calculated by using the frequency of the term.
3. **Query construction.** To calculate the similarity between the source code and bug report, the bug report is also preprocessed, similar to the corpus generation step.
4. **Ranking.** The indexed corpus and the bug report are transformed into vectors. In the case of the vector space model (VSM) approach, we can obtain their similarity by calculating the cosine similarity of two vectors. The calculated similarity values are then used to rank the modules. The higher the rank, the more likely it is to be the location of a bug.

An advantage of IR-based bug localization techniques is that a few types of inputs are required. As most software development projects currently use an issue tracking system, the bug localization techniques can easily obtain their source code and bug reports. Therefore, IR-based bug localization can be applied in most situations. In contrast, a disadvantage is their low accuracy; sufficient accuracy cannot be obtained by merely considering the similarity between the bug report and source code (Zhou et al., 2012). In addition, this approach depends on the quality of bug reports (Chaparro et al., 2017; Kim et al., 2013; Le et al., 2017).

Because the accuracy of IR-based bug localization is not sufficiently high, many approaches have been proposed to combine it with other information types. It is noteworthy that the similarity between the bug report and source code is still necessary. Other types of information are solely an addition. Types of information that have been applied to bug localization are as follows.

- **Source code size.** Zhou et al. (2012) used the number of lines of source code to represent the size of the source code because bugs are likely to be in the large size source code.
- **Past bug reports.** Zhou et al. (2012) used bug reports similar to current bug reports to support their approach. The underlying reason is that similar bug reports are likely to modify the same file.
- **Stack trace.** Wong et al. (2014) used stack trace information in a bug report to capture the order of the executed modules until the program failed. Therefore, as we know the modules that were executed, we can use such information to improve bug localization.
- **Change history.** Wang and Lo (2014) used the past change history when calculating the probability of buggy modules. This information represents the likelihood that a given file will contain a bug in general. This information can be calculated by the number of modified files in each bug fixing commit.

The advantages and disadvantages of the approaches that combine other information can be the opposite of IR-based approaches. Because information such as the change history is not always available, the applicability of some approaches is limited. In contrast, as the approaches use extra information, the accuracy can be improved; for instance, an approach in which the combined size of the source code can significantly improve the IR-based bug localization accuracy (Zhou et al., 2012).

2.2. Code smells

Code smells are often used as an indicator of a design flaw or problem in the source code (Fowler, 1999). Many studies have found that code smells are related to several aspects of software development problems (Yamashita and Moonen, 2012, 2013). Thus, it is recommended to remove code smells by performing related refactoring operations to improve the quality of the source code. Code smells were initially proposed using descriptive language. Thus, several studies have attempted to implement them formally.

For example, Lanza and Marinescu (2007) defined a metric-based strategy for detecting God Class as follows:

$$\begin{aligned}\text{God Class}(m) = {} & (V_{\mathrm{ATFD}}(m) \geq T_{\mathrm{ATFD}}) \wedge \\ & (V_{\mathrm{WMC}}(m) \geq T_{\mathrm{WMC}}) \wedge \\ & (V_{\mathrm{TCC}}(m) \leq T_{\mathrm{TCC}})\end{aligned}$$

where $V_{\mathrm{ATFD}}(m)$, $V_{\mathrm{WMC}}(m)$, and $V_{\mathrm{TCC}}(m)$ are the metric values of access to foreign data (ATFD), weighted method count (WMC), and tight capsule cohesion (TCC) of m, respectively. Similarly, T_{ATFD}, T_{WMC}, and T_{TCC} are the thresholds of ATFD, WMC, and TCC, respectively. Specifically, ATFD measures the number of foreign attributes that are sued by a class. Therefore, the higher the ATFD, the more likely the class is to be a God Class. Similarly, WMC represents the sum of the complexities of all the methods declared in a class. Thus, the higher the WMC, the more likely it is that the class is a God Class. In contrast, the TCC represents the degree of cohesiveness of a class. As a result, a class with a lower TCC is more likely to be a God Class. These three conditions are then combined as a conjunctive form to determine God Class.

To measure the strength of a code smell, Marinescu defined *severity* as an integer that measures the number of times the value of a chosen metric exceeds a given threshold (Marinescu, 2012). For instance, in the case of God Class, WMC, ATFD, and TCC were used in the detection approach. Among these three metrics, ATFD is used to calculate the severity. In other words, we can calculate the severity by computing the number of times the value of ATFD exceeds its threshold. Severity values range from 1 to 10. Note that the metrics used to compute the severity vary according to the smell type because the detection strategy of each smell type uses a different set of metrics. More information, including additional examples of the severity computation, can be found in the original paper by Marinescu (2012).

3. Related work

IR-based bug localization is useful for locating source code files that need to be modified to fix a specific bug. If a bug report and the source code are its inputs, it outputs files or a ranking of files that need to be modified to fix the bug. IR techniques, e.g., Latent Dirichlet Allocation (LDA) (Blei et al., 2003), Latent Semantic Indexing (LSI) (Landauer et al., 1998), or Vector Space Model (VSM) (Salton et al., 1975), are used for bug localization to calculate the similarity between a given bug report and the source code. The obtained similarity scores were then utilized to specify the source files from the given bug report. In addition, techniques of IR-based concept location (Gay et al., 2009; Dit et al., 2013) and impact analysis (Gethers et al., 2012) follow the same approach. Among them, Rao and Kak reported VSM to be the best choice among IR techniques for bug localization (Rao and Kak, 2011).

An advantage of IR-based bug localization techniques is the few types of inputs that need to be prepared because they require only bug reports and source code as their inputs. However, this could also be regarded as a disadvantage because they depend excessively on the quality of bug reports. In other words, IR techniques are unable to process low-quality bug reports effectively. To mitigate this problem, researchers focused directly on the quality of bug descriptions and the tailoring of IR techniques. Chaparro et al. (2017) proposed a technique to improve IR-based bug localization by reconstructing low-quality bug reports. Other researchers attempted to rectify the behavior of an IR-based bug localization technique according to the quality of bug reports (Kim et al., 2013; Le et al., 2017). Also, Moreno et al. (2015) proposed a technique named QUEST to automatically configuring the parameters of the used IR approach to improve its accuracy.

Another line of techniques involves the combination of additional information with an IR technique. Shi et al. (2018) suggested that combining additional information with an IR-based bug localization technique can be beneficial. Zhou et al. (2012) defined revised VSM (rVSM) by considering the scale of the source code in VSM. They proposed BugLocator by combining rVSM with information about similar bug reports from the past. Wong et al. (2014) proposed BRTracer by combining it with stack traces. A stack trace describes the methods that are invoked and the order in which they are invoked until the test fails, which can be obtained by dynamic analysis or reported in bug reports. Similarly to BRTracer, Lobster (Moreno et al., 2014) also utilized stack trace information in bug reports to improve bug localization. Tantithamthavorn et al. (2013) proposed a technique that uses the history of past changes together with BugLocator. In this technique, they used the co-change information in the change history and additionally specified modules that were likely to be changed when a module was changed. Furthermore, BLUiR (Saha et al., 2013) considered the structure of the source code in addition to BugLocator. AmaLgam (Wang and Lo, 2014) utilizes a bug prediction technique using the version history in addition to BLUiR.

Although these hybrid IR-based bug localization techniques are more accurate than basic IR-based bug localization techniques, they are more costly to apply. For instance, AmaLgam requires the user to collect the change history, which is time-consuming. In addition, the applicability of techniques using a version history is limited to projects with sufficient history.

This study extends our previous study using Bench4BL, which is the largest available dataset for bug localization. The techniques from Bench4BL that we used in this study are BugLocator, BRTracer, BLUiR, and AmaLgam. In addition, we added VSM as a baseline for bug localization techniques and rVSM as a technique that considers the size of the source code. The comparison of the information used in each technique is presented in Table 1. As we can see, most techniques are improved based on another technique by adding more information. In contrast, our smell-aware technique is dependent on another technique and can be applied on any technique to improve the performance.

As we have already noted in Section 1, the motivation behind our approach was that modules with code smells have been found to be more changed- and fault-prone (Khomh et al., 2012; Guerrouj et al., 2017). Following a similar motivation, there are several attempts to use smell information for bug prediction. For example, Taba et al. (2013) have proposed a bug prediction technique that uses a historical metric computed from smelly classes as additional information. In addition, Palomba et al. (2017) have succeeded in effectively utilizing the severity degree of code smells in bug prediction. Their studies suggest the usefulness of smell-based information on identifying buggy portion in source code, which partly justify the use of smell-based information on bug localization.

4. Smell-Aware bug localization

4.1. Bug likelihood index

The smell-aware bug localization technique aims to improve the accuracy of existing IR-based bug localization. A problem with IR-based bug localization is that it relies on the textual similarity between a bug report and the source code, i.e., it considers all modules to be equal and does not consider the likelihood of a module containing a bug. This shortcoming may be responsible for the low accuracy of the technique. To overcome this problem, our technique uses information about the code smells to represent the likelihood of a module containing a bug. Specifically, we used the *smell severity*, which indicates the strength of a code smell (Marinescu, 2012), to represent the likelihood. In addition, as smell information can be directly detected from the source code, it is possible to keep the cost of the technique close to that of IR-based bug localization. In other words, the user does not need to obtain additional information to use the technique.

Example. To fix the bug CAMEL-9059,[1] the method `createEndpoint` in the class `JettyHttpComponent` was modified.[2] The class had three smells: Refused Parent Bequest,

[1] https://issues.apache.org/jira/browse/CAMEL-9059

[2] https://github.com/apache/camel/commit/a516501

Table 1
Comparison of existing techniques.

Information used	VSM	rVSM	BugLocator	BRTracer	BLUiR	AmaLgam	Smell-aware
Similarity	✓	✓	✓	✓	✓	✓	
Code size		✓	✓	✓	✓	✓	
Past bug reports			✓	✓	✓	✓	
Stack trace				✓			
Code structure					✓	✓	
Change history						✓	
Smell information							✓

Schizophrenic Class, and God Class . Also, the method had two smells: Feature Envy and Blob Operation . In particular, God Class and Feature Envy were detected as the severest (severity of 10). This class module is highly complicated, difficult to comprehend, and difficult to change safely, which might lead to bug-proneness. Unfortunately, this module was assigned a low rank in IR-based bug localization (349th in the ranking by VSM) because many other modules were textually more similar to the bug report. Smell-aware bug localization aims to increase the ranking of these modules by utilizing smell information.

To combine the code smell information with textual similarity, we proposed the *bug likelihood index* (BLI). The BLI for each module m can be calculated as follows:

$$BLI(m) = (1-\alpha)\, nSim(m) + \alpha\, nSev(m),$$

where $nSim(m)$ is the textual similarity of the bug report and module m based on VSM and $nSev(m)$ is the sum of the severity of the smells contained in module m. Both values are normalized in the range of [0, 1]. More specifically, they are calculated from the original non-normalized similarity $Sim(m)$ and severity sum $Sev(m)$ as follows:

$$nSim(m) = \frac{Sim(m)}{\max_{m' \in M} Sim(m')},$$

$$nSev(m) = \frac{Sev(m)}{\max_{m' \in M} Sev(m')}.$$

Further, α $(0 \le \alpha \le 1)$ is a parameter representing the weight of $nSim(m)$ and $nSev(m)$.

In our previous study, we conducted an experiment using four open-source projects: ArgoUML, JabRef, jEdit, and muCommander. In this study, TraceLab (Dit et al., 2012) and inFusion (Marinescu, 2012) were used as a VSM implementation and a smell detector, respectively. In the VSM implementation of TraceLab, TF–IDF was used as its weighting scheme, and it was equipped with a standard IR preprocess. We applied the IR-based bug localization technique with VSM and our smell-aware bug localization to the targets and compared the accuracy using the mean average precision (MAP) (Cormack and Lynam, 2006). As a result, our technique could improve IR-based bug localization by 36%, 34%, 24%, and 28% in relative comparison for ArgoUML, JabRef, jEdit, and muCommander, respectively.

4.2. Generalized bug likelihood index

Although the result using BLI was promising, the following limitations must be noted.

- **Generalization.** As the goal of our previous study was to obtain a preliminary result to determine whether code smells have the potential to improve bug localization, we conducted our study on a small dataset consisting of four projects. Although this dataset is often used in related studies reported in the literature, it is difficult to generalize the applicability of our technique because of the small number of projects.
- **Optimal configuration.** The generalization of smell-awareness requires many parameters to be specified. For example, we could specify the granularity of code smells, e.g., class or method level. We could also specify the aggregator to use when a module has more than one code smell, e.g., summation or obtaining the maximum. Finally, as many types of code smells exist, we could choose whether to include all types or only specific types of code smells. However, in our previous work, we conducted a study with only one configuration, which means that the reported results might not be optimal.
- **Combination of different base techniques.** Smell-aware bug localization was designed to be employed in combination with a base bug localization technique to improve its performance. However, in our previous study, VSM was the only base technique to be studied. Although VSM is a representative technique for IR-based bug localization because of its simplicity, many other high-performance bug localization techniques have been proposed (Wong et al., 2014; Zhou et al., 2012; Saha et al., 2013; Wang and Lo, 2014). Therefore, it remained unclear whether our technique can be used to improve other base techniques.

Therefore, to overcome the limitations mentioned above, we generalized the technique to examine different configurations of code smells and defined *generalized bug localization index* (gBLI). The gBLI of module m can be calculated as:

$$gBLI^{t,c}(m) = (1-\alpha)\, nScore^{t}(m) + \alpha\, nSmell^{c}(m)$$

where $nScore^{t}(m)$ is the normalized output score of the bug localization technique, t. Furthermore, $nSmell^{c}(m)$ is a normalized value based on the code smell configuration c, which includes three parameters: granularity (g), aggregator (a), and type selector (s). The normalization process of $nScore(m)$ and $nSmell^{c}(m)$ is the same as that of $nSim(m)$ and $nSev(m)$. The $nScore^{t}(m)$ and $nSmell^{c}(m)$ are generalizations of $nSim(m)$ and $nSev(m)$, respectively. The details of the code smell configuration will be explained in the next subsection.

4.3. Code smell configuration

A code smell configuration includes three parameters: granularity (g), aggregator (a), and type selector (s), which are explained in the following paragraphs.

Granularity (g). Code smells are often defined on the basis of the granularity of the modules, e.g., the class or method levels. In our previous study, we used only class-level code smells because we focused on a bug localization technique that outputs class-level results. In other words, we kept the granularity of the code smells the same as that of the bug localization result. However, in addition to the class-level code smells, adding method-level smells may improve the performance because these smells add more information to the modules. In addition, we are likely to obtain a larger number of modules with code smells by considering the method-level code smells. Nevertheless, this information may add to the noise of the technique and decrease its performance.

Therefore, it might be useful to clarify the effect of using different code smell granularities when applying the technique.

Therefore, in addition to class-level code smells, the use of method-level code smells may improve the performance. The granularity (g) can be set as follows:

- g_1: *class level*,
- g_2: *method level*, and
- g_3: *both class and method levels*.

Aggregator (a). When detecting code smells, the possibility of more than one code smell being detected in a single module is high. Therefore, we need an aggregator to combine the information of each code smell to represent the value of the module. For example, in our previous study, we used summation to combine the severity of each smell in a module. As another example, Palomba et al. (2017) used the maximum smell severity value to extract the bug-proneness of a module. Thus, multiple ways are available in which to combine the smell information. A comparison of different aggregators is necessary to determine which one performs the best.

We considered the sum and maximum of the severity as aggregators because it was used in the literature. In addition, we considered using the smell existence, that is, 1 if a module contains at least one smell and 0 otherwise. We also considered the number of smells in a module. Furthermore, the average and the median of the severity of all the smells are considered as representatives of the severity degree of the target module. Finally, considering a situation in which the severity or number of smells is biased depending on the smell types, nested aggregators using the average or median after aggregating by the maximum severity or number of smells are also added. These aggregators are considered to confirm whether the use of the smell severity yields improved performance. To summarize, the aggregator (a) can be set as follows:

- a_1: *sum of severity*,
- a_2: *maximum severity*,
- a_3: *existence of smells*,
- a_4: *number of smells*,
- a_5: *average of severity*,
- a_6: *median of severity*,
- a_7: *average of the maximum severity in each smell type*,
- a_8: *median of the maximum severity in each smell type*,
- a_9: *average number of smells in each smell type*, and
- a_{10}: *median number of smells in each smell type*.

Type Selector (s). Most detectors can detect different types of code smells; for example, inFusion can detect 16 types of code smells. Our previous study entailed the detection of all types of code smells. Nevertheless, different types of code smells may affect bug-proneness differently. Therefore, employing a technique with different types of code smells might yield different results.

Our goal here is to compare the performance when all types of code smells are used with the performance when only certain types of code smells that are more likely to be related to bug proneness are used. We prepare five setting levels of smell type selection:

- s_1: *all smell types*,
- s_2: *rare selected smell types*,
- s_3: *medium rare selected smell types*,
- s_4: *medium selected smell types*, and
- s_5: *well selected smell types*,

which vary on their different inclusiveness of smell types. The concrete selection of each selector is open at this stage because they will be determined according to an empirical study; see Sections 5.6 and 5.7 .

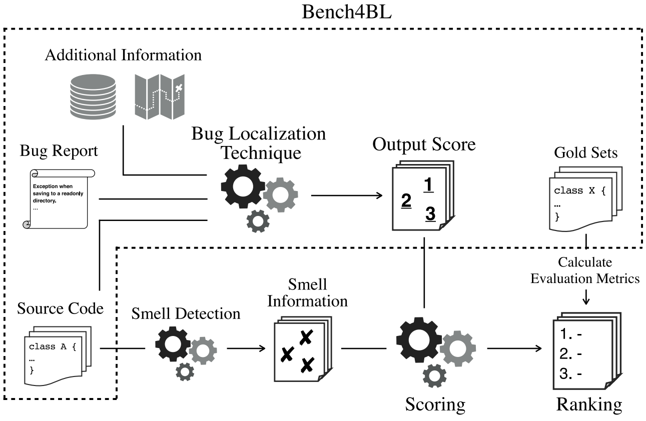


Fig. 1. Overview of this study.

Specialization to BLI. Because the previous technique for class modules used VSM as its base bug localization technique and the a_1: sum of the severity of g_1: class-level smells of s_1: all smell types as its configuration of code smells, gBLI with this configuration can express our original BLI:

$$gBLI^{\mathsf{VSM},\langle g_1,a_1,s_1\rangle}(m) \equiv BLI(m).$$

5. Empirical study

5.1. Research questions

This study focuses on the following four research questions (RQs).

- RQ_1: *Does smell-aware bug localization improve IR-based bug localization using VSM even for a large-scale dataset?*
- RQ_2: *What is the relationship between the performance improvement of the smell-aware bug localization and bug proneness?*
- RQ_3: *What are the best configurations for smell-aware bug localization as an extension of VSM?*
- RQ_4: *Is the performance of smell-aware bug localization superior to that of state-of-the-art bug localization techniques?*

Details of the motivation for each respective RQ are provided later.

5.2. Approach overview

An overview of this study, including the process of Bench4BL and smell detection, is shown schematically in Fig. 1. The nodes within the dotted enclosure are provided by Bench4BL. First, we executed the bug localization technique using the source code, bug reports, and additional information as inputs. Next, we detected the code smells from the source code using the aforementioned tool. Then, we calculated the score of each module from the smell information and output the score obtained by the bug localization technique. The scores are then used to generate the output ranking. Finally, we calculated the accuracy of the ranking based on the gold set included in Bench4BL.

5.3. Data collection

5.3.1. Bench4BL

In this study, we used Bench4BL, which is the largest benchmark for bug localization (Lee et al., 2018). The dataset contains the source code, bug reports, and lists of source files that were modified to fix the bug reports, i.e., gold sets, across 46 projects and their versions. Bench4BL also includes the implementation of state-of-the-art bug localization techniques such as BugLocator (Zhou et al., 2012), BRTracer (Wong et al., 2014), BLUiR (Saha et al., 2013), AmaLgam (Wang and Lo, 2014), BLIA (Youm et al., 2017), and Locus (Wen et al., 2016). We decided to use Bench4BL because it is suitable for conducting a large-scale empirical study. Furthermore, it would enable us to combine our smell-aware bug localization with the base techniques implemented in Bench4BL.

We regarded each project version in Bench4BL as a specific system. In a typical bug localization context, we see a system as a pair of (1) a set of bug reports that define bugs and (2) a source code snapshot to be used to locate the bugs. When applying this approach to Bench4BL, project versions are most suitable for systems because bug reports are associated with a project version in Bench4BL. This decision also means that we regard different versions of the same project as different systems.

5.3.2. Bug localization techniques

Of the six bug localization techniques (BugLocator, BRTracer, BLUiR, AmaLgam, BLIA, and Locus) provided by Bench4BL, we excluded BLIA and Locus and used the left four techniques for two reasons. First, these two implementations often behave non-deterministically, outputting different results from the same input. This behavior was not suitable for our study. Second, these two implementations had more invalid outputs than the other four implementations. Following the bug report selection and invalidity criteria shown in Section 5.3.4, we could collect 6936 bug reports that met our criteria from the VSM results. If we used the four techniques as mentioned above, the number of bug reports decreased to 6931, which means that only five bug reports were excluded. However, if we added BLIA into these four, 199 reports were additionally excluded, and 6732 bug reports remained. If we added Locus in addition to BLIA, 752 reports were additionally excluded, and 5980 remained. Since our goal was to reproducibly confirm the effectiveness of the smell-aware approach on the improved bug localization techniques, not necessarily on all bug localization techniques, we excluded BLIA and Locus to include more bug reports in the experiment. A similar buggy behavior of the Locus implementation in Bench4BL was also reported by Chaparro et al. (2019). However, note that the second reason does not directly imply that these implementations are broken. Some results are valid in terms of bug localization results but inappropriate for our study; see Section 5.3.4 for the details.

In addition to the four bug localization techniques provided by Bench4BL, we added two techniques: VSM and rVSM. Both of these techniques are components used in BugLocator, and we used them to ensure that the implementations were compliant with the Bench4BL framework. VSM is the basis of IR-based bug localization and computes the textual similarity between the bug report and the source code. We added this technique because it has the lowest cost among bug localization techniques. We also added rVSM, an extension of VSM, because of our intention to confirm whether smell-aware bug localization is effective only because it considers the size of the source code. Although we mentioned that code smell is effective for bug localization, the underlying reason may be the size of the source code because certain information from smells represents information about the size of the source code. Therefore, we compare the smell-aware technique with rVSM, which considers the size of the source code in RQ_4.

To summarize, we selected the six bug localization techniques shown in Fig. 2. Because all the techniques are based on VSM, we can compare the effects of the additional information contributed by each technique.

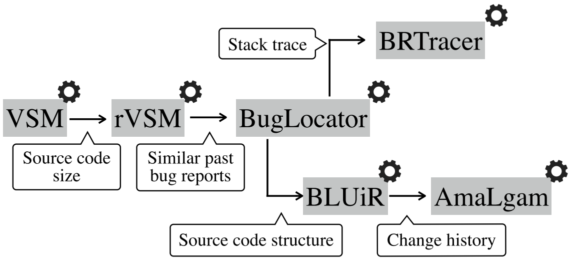


Fig. 2. Bug localization techniques used in this study.

- VSM (Zhou et al., 2012): The vector space model technique used in BugLocator.
- rVSM (Zhou et al., 2012): Extension of VSM that considers the size of the source code.
- BugLocator (Zhou et al., 2012): A technique that combines rVSM with previous similar bug reports.
- BRTracer (Wong et al., 2014): A technique that combines BugLocator with extracted stack traces.
- BLUiR (Saha et al., 2013): A technique that combines BugLocator with structural information obtained from the source code.
- AmaLgam (Wang and Lo, 2014): A technique that combines BLUiR with historical information.

5.3.3. Code smell detection

In this study, code smells were detected from the source code of each version in Bench4BL. We used inFusion (Marinescu, 2012)[3], a powerful commercial code smell detector. The inFusion is an extended version of inCode (Ganea et al., 2017), which is a successor to iPlasma (Marinescu et al., 2005). We selected inFusion for several reasons:

- the detected smell instances are associated with a severity score,
- it can be assembled in an automated manner without requiring the collection of dependent libraries, compilation of source code, and configuration on an IDE, which helps to simplify the workflow of our experiment with less manual effort,
- it follows well-known metric-based smell detection strategies (Lanza and Marinescu, 2007), which are explained in Section 2.2, and
- it can detect 16 types of code smells at both the class and the method level, which suits our need considering that we aim to compare the effect of different types of code smells in our approach.

We regarded the first two reasons as mandatory requirements to conduct our study. Although other code smell detectors such as cASpER (Stefano et al., 2020), DECOR (Moha et al., 2010), JCodeOdor (Fontana et al., 2015b), and JDeodorant (Tsantalis et al., 2008), have been proposed to date, they have not met these requirements.

The class-level and method-level smells detected by inFusion are summarized in Table 2. We used the detection result of both of these types of smells without manual validation.

[3] Because the production company has been closed, it is no longer available. The detected smell instances used in this study are enclosed in the online appendix (Takahashi et al., 2021).

Table 2
Code smells detected by inFusion.

Granularity	Types	Description
Class level	Blob Class Fowler (1999), Lanza and Marinescu (2007), Brown et al. (1998)	A class that is very large and complex
	Data Class Fowler (1999), Lanza and Marinescu (2007), Riel (1996)	A class with no functionality, only data
	Distorted Hierarchy Riel (1996)	A class with very narrow and deep inheritance hierarchies
	God Class Fowler (1999), Lanza and Marinescu (2007), Riel (1996)	A class that handles data from other classes
	Refused Parent Bequest Lanza and Marinescu (2007), Riel (1996), Martin (2007)	A class that rarely uses members that inherit from its base class
	Schizophrenic Class Riel (1996), Martin (2007)	A class representing multiple concepts
	Tradition Breaker Lanza and Marinescu (2007), Riel (1996)	A class that violates the conventions defined by its base class
Method level	Blob Operation Fowler (1999), Lanza and Marinescu (2007), Brown et al. (1998)	A method that is large and complex
	Data Clumps Fowler (1999)	A method in which several data values appear as a group
	External Duplication Fowler (1999), Brown et al. (1998), Hunt and Thomas (2000)	A method containing duplicate code with unrelated classes
	Feature Envy Fowler (1999), Lanza and Marinescu (2007), Riel (1996)	A method that is more relevant to the data of other classes than those of its own class
	Intensive Coupling Fowler (1999), Lanza and Marinescu (2007), Riel (1996)	A method that is more associative with many other methods
	Internal Duplication Fowler (1999), Brown et al. (1998), Hunt and Thomas (2000)	A method with duplicate code in its own class
	Message Chains Fowler (1999), Lanza and Marinescu (2007), Brown et al. (1998)	A method that results in a chain of many method calls
	Shotgun Surgery Fowler (1999), Lanza and Marinescu (2007)	A method that propagates changes to many other methods when it is changed
	Sibling Duplication Fowler (1999), Brown et al. (1998), Hunt and Thomas (2000)	A method with duplicate codes between sibling classes

5.3.4. Data selection

We excluded data with any inconsistency from the dataset. For example, output rankings may be invalid, e.g., those including the similarity score value of Not-a-Number (NaN) or those with no gold module occurrences. Although the rankings with no gold module occurrences are valid as bug localization results, we regarded them as invalid because they are not useful for our study in terms of confirming the improvements of the smell-aware bug localization approach. We excluded any bug reports where the output ranking of at least one bug localization technique was invalid. In addition, we excluded bug reports of the versions for which inFusion could not detect any code smells because of our intention to confirm the effectiveness of the smell-aware technique. Finally, we excluded versions with fewer than five bug reports to ensure that each version involved a certain minimum number of bug reports. This approach was necessary to mitigate the threat of over-optimizing the α values in the case of a small number of bug reports; see Sections 5.5.2 and 6.1 for the details. As a result, we excluded 2528 of the 9459 bug reports and used 6931 bug reports over 309 versions and 35 projects. Table 3 provides information about each of these projects. The columns in this table contain the name of the project, the number of versions, the total number of bug reports, and the average number of source files.

Table 3
Projects used.

Group	Project	# versions	# bug reports	# files (mean)
Apache	CAMEL	39	1390	9576
	HBASE	27	682	2203
	HIVE	21	1215	2777
	CODEC	4	32	75
	COLLECTIONS	3	70	468
	COMPRESS	11	103	174
	CRYPTO	1	6	82
	CSV	2	12	26
	IO	7	78	156
	LANG	10	169	241
	MATH	10	191	781
JBoss	ENTESB	1	12	252
	JBMETA	1	15	784
	SWARM	2	35	493
	WFCORE	8	338	3377
Spring	AMQP	6	50	266
	ANDROID	1	10	179
	BATCH	17	355	1180
	BATCHADM	2	16	215
	DATACMNS	12	93	352
	DATAGRAPH	2	12	501
	DATAJPA	11	66	222
	DATAMONGO	19	206	418
	DATAREDIS	3	21	331
	DATAREST	10	91	271
	LDAP	1	42	433
	MOBILE	1	7	62
	ROO	11	687	681
	SEC	27	446	1038
	SECOAUTH	1	68	409
	SGF	7	60	274
	SHDP	2	28	956
	SHL	1	8	89
	SPR	5	111	4162
	SWF	9	92	668
	SWS	14	114	756
	Total	309	6931	2119

5.4. Evaluation metrics

To evaluate the ranking outputs from each bug localization technique, we used the following evaluation metrics:

- ***Top N*.** This metric represents the ratio in which at least one gold file is included within the top N of the given ranking. Here, *gold files* denote files included in the gold set. Given a set of bug reports B, the metric can be calculated as follows:

  $$Top\ N = \frac{1}{|B|}\sum_{b \in B} \mathrm{top}_N(b)$$

  where $\mathrm{top}_N(b)$ returns 1 if a gold file is contained in the top N of the ranking obtained from bug report b, and 0 otherwise. By definition, we can calculate the actual number of bug reports that succeeded in having at least one gold module within the top N of the given ranking by multiplying the Top N value by the total number of bug reports. In this study, we used Top 1, Top 5, and Top 10.

- **Mean Reciprocal Rank (MRR).** MRR (Craswell, 2009) is the mean of the multiplicative inverse of the rank of the first gold file in the given ranking. Given a bug report b, its reciprocal rank (RR) can be calculated as follows:

$$RR(b) = \frac{1}{\text{rank}(b)}$$

where rank(b) is the ranking of the highest gold file in the ranking obtained from bug report b. Given a set of bug reports B, the MRR is calculated as the average of the RR of each bug report in B:

$$MRR = \frac{1}{|B|}\sum_{b \in B} RR(b).$$

- **Mean Average Precision (MAP).** MAP (Cormack and Lynam, 2006) considers all the gold files, whereas Top N and MRR consider only the top gold files. Assuming that the number of files in the output ranking is N, the average precision (AP) of a bug report b can be calculated as follows:

$$AP(b) = \sum_{i=1}^{N} \frac{precision_b(i) \times gold_b(i)}{\text{number of gold files}}.$$

Here, i denotes the rank of a file and $precision_b(i)$ denotes the ratio of gold files in the files ranked at or higher than the ith rank. $gold_b(i)$ returns 1 if the file ranked at the ith position is in the gold set and 0 otherwise. MAP is the average of AP of all the bug reports B:

$$MAP = \frac{1}{|B|}\sum_{b \in B} AP(b).$$

Moreover, we used the Wilcoxon signed-rank test (Wilcoxon et al., 1970) for statistical testing. Because the values of all the evaluation metrics are computed as the average of the values for each bug report, we used the set of values for each bug report for all the statistical tests in this study. For example, when testing the statistical significance of the difference between the two techniques in terms of MAP aspect, we compared two sets of AP values, which were used to compute MAP. In addition, when reporting the statistical significance, we also reported Cliff's delta (d) as a measure of the magnitude of the improvement. The Cliff's delta is interpreted based on the threshold by Romano et al. (2006): *negligible* for $|d| < 0.147$, *small* for $0.147 \leq |d| < 0.33$, *medium* for $0.33 \leq |d| < 0.474$, and *large* for $0.474 \leq |d|$.

5.5. RQ$_1$: Does smell-aware bug localization improve IR-based bug localization using VSM even for a large-scale dataset?

5.5.1. Motivation

Although our previous study showed that smell-aware bug localization can significantly improve the accuracy of IR-based bug localization, the study was conducted with only four projects. In addition, the number of bug reports included in the dataset was only 277, which made it difficult to generalize the results. Therefore, the goal of this RQ is to verify whether the smell-aware bug localization technique can be used to improve the performance even for a large-scale dataset. This is intended as a sanity check to confirm whether the same setting involving the use of the original smell-aware bug localization would be applicable to systems in Bench4BL prior to making new attempts.

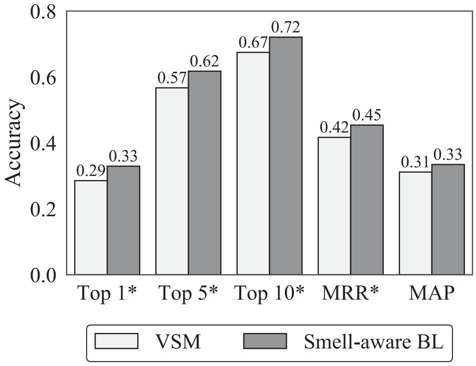


Fig. 3. Comparison of the accuracy of smell-aware bug localization and VSM.

5.5.2. Study design

This study was designed to replicate and extend the original study to all projects in Bench4BL. The smell-aware bug localization technique can be employed by setting the smell granularity to either the class or method level. We used the smell granularity at the class level because the granularity of modules obtained in bug localization is at the file level in Bench4BL. In addition, we used a Java file as a proxy for the class and excluded all inner classes. The α value was set to a value such that each evaluation metric is maximized in each system, which ensures that the study is conducted under the same conditions as in the previous experiment. We produced the rankings for each system by calculating the BLIs from the bug reports in the system using all the α possibilities, which range from 0 to 1 in increments of 0.01. We then evaluated the set of rankings for each α by employing the evaluation metric that was used. Finally, the α value that maximized the evaluation score was used as the parameter for the pair of the evaluation metric and the system.

For each bug report, all files in the bug localization result were sorted in descending order of BLI, and the accuracy of IR-based bug localization and the smell-aware bug localization technique were compared according to the gold set in Bench4BL.

5.5.3. Results

The results are shown in Fig. 3. Clearly, the bug localization results for all the evaluation metrics improved. The increase ratios in terms of the Top 1, Top 5, Top 10, MRR, and MAP metrics were 15.4%, 8.9%, 6.9%, 8.9%, and 7.3% in relative comparison (0.044, 0.050, 0.046, 0.037, and 0.023 in absolute comparison), respectively. The metrics that are statistically significant ($p < 0.01$) are denoted by an asterisk (*). All the differences were statistically significant except for the improvement in MAP: $p = 0.13$ (Cliff's delta: 0.044, 0.050, 0.046, 0.051, and 0.042, all *negligible*). For example, Top 10 increased by 0.046 (from 0.675 to 0.721), which means that the total number of bug reports with gold modules in their top 10-ranked items increased by 321 (from 4676 to 4997).

It is noteworthy that MAP in this study increased by a lesser amount than in the previous study. Specifically, in this study, MAP increased by 7.3% on average in relative comparison, whereas in the previous study, it was 30.5%. The difference may be attributed to several factors. First, the previous experiment involved all the classes, including the inner classes, whereas this experiment considered only the top-level classes and excluded the inner classes. Next, the VSM implementation we used differed from that in the previous study. The VSM implementation in this study is the VSM part of BugLocator, which was optimized for bug localization usage (Zhou et al., 2012). In addition, the number of projects and bug reports used in this study is much larger than in the previous study; i.e., 4 vs. 36 projects and 277 vs. 6943 bug reports were

Table 4
Application example to HBASE-1795.

Rank	Module	*nScore*	*nSmell*	*BLI*
(a) VSM ($\alpha = 0$)				
1	`TestTHLog`	1.000	0.000	1.000
2	`THLogRecoveryManager`	0.949	0.000	0.949
3	`HLog`	0.792	0.231	0.792
4	`TransactionalRegion`	0.784	0.000	0.784
5	`TestStoreScanner`	0.659	0.000	0.659
6	`HRegion`	0.642	0.769	0.642
7	`BatchMutation`	0.637	0.000	0.637
8	`TestTHLogRecovery`	0.619	0.000	0.619
9	`TestMinorCompactingStoreScanner`	0.600	0.000	0.600
10	`Store`	0.582	1.000	0.582
(b) Smell-aware approach ($\alpha = 0.31$)				
Rank	Module	*nScore*	*nSmell*	*BLI*
1	`Store`	0.582	1.000	0.711
2	`TestTHLog`	1.000	0.000	0.690
3	`HRegion`	0.642	0.769	0.681
4	`THLogRecoveryManager`	0.949	0.000	0.655
5	`HLog`	0.792	0.231	0.618
6	`TransactionalRegion`	0.784	0.000	0.541
7	`ServerManager`	0.364	0.769	0.489
8	`IndexedRegion`	0.325	0.769	0.462
9	`TestStoreScanner`	0.659	0.000	0.455
10	`BatchMutation`	0.637	0.000	0.440

used previously and in this study, respectively. Consequently, the result of our previous study might be an extreme case, whereas the results of this study reflect a more realistic distribution.

Example. The application of smell-aware bug localization to HBASE-1795[4] is presented in Table 4. In Table 4, the top 10 items of the result using VSM ($\alpha = 0$) and that using the smell-aware bug localization technique ($\alpha = 0.31$) are compared. The gold module, i.e., the class module that was modified to fix this bug, is highlighted in gray. The gold module `Store`, which is located at the tenth rank in VSM, has several class-level smells such as Blob Class and God Class. Therefore, `Store` eventually had the top *nSmell* score and was ranked at the top when using smell-aware bug localization. In addition to HBASE-1795, a total of 24 bug reports tied to HBASE 0.20.5 targeted `Store` to fix, and the score improvement of `Store` led to improved rankings for many in the system. Note that in addition to the class-level smells, certain methods of `Store` have a method-level smell called Blob Operation, which could also be utilized in smell-aware bug localization. The use of such method-level smells was studied in RQ_3.

In summary, the smell-aware bug localization technique at the class level can also improve the accuracy of IR-based bug localization using VSM even for a large dataset.

5.6. RQ$_2$: What is the relationship between the performance improvement of the smell-aware bug localization and bug proneness?

5.6.1. Motivation

We showed that smell-aware bug localization could also improve the bug localization accuracy even for the Bench4BL dataset in answering RQ_1. As already explained in Section 1, we consider that the results of smell-aware bug localization are improved because smells are bug prone (Khomh et al., 2012; Guerrouj et al., 2017). To provide more convincing evidence of bug-proneness in the bug localization context in this study and investigate the difference in contributions by the smell types, we examine the extent to which each smell type affects the possibility of identifying buggy modules by bug localization.

5.6.2. Study design

To answer RQ_2, we calculate the *relative risk* (Sistrom and Garvan, 2004) of the existence of buggy modules for each smell type. Let M_{all} and $B_{\mathrm{all}} \subseteq M_{\mathrm{all}}$ be the sets of all the modules and the *buggy* modules for all the target 309 project versions, where each element is represented as a pair of a module name and the project version to which it belongs. Here, we regarded a module as buggy if and only if the module was included in the gold set of at least one bug report in the target version in Bench4BL. We denote the set of modules in which a smell of type t is detected by $M_t \subseteq M_{\mathrm{all}}$. The set of buggy modules that contains the smell of t is computed as $B_t = M_t \cap B_{\mathrm{all}}$. Then, the risk of smelly modules that are likely to contain bugs to be fixed (Risk_t), that of non-smelly modules (Risk^*_t), and the relative risk of smelly modules (RR_t) are, respectively, expressed as follows:

$$\mathrm{Risk}_t = \frac{|B_t|}{|M_t|}, \mathrm{Risk}^*_t = \frac{|B_{\mathrm{all}} \setminus B_t|}{|M_{\mathrm{all}} \setminus M_t|}, \mathrm{RR}_t = \frac{\mathrm{Risk}_t}{\mathrm{Risk}^*_t}.$$

5.6.3. Results

The results are presented in Table 5. The columns show the smell type, the number of detected smelly modules ($|M_t|$), the number of buggy modules in the files detected as smelly ($|B_t|$), the buggy risk of the smelly modules (Risk_t) and non-smelly modules (Risk^*_t), and the buggy relative risk (RR_t) for all systems. The smell types are sorted in descending order of the obtained relative risks of the smelly modules. The total number of modules ($|M_{\mathrm{all}}| = 654674$) and buggy modules ($|B_{\mathrm{all}}| = 15834$) are also shown in the bottom row of the table. The results in the table indicate that, when considering all smell types, 5.735% of the smelly modules are buggy modules. The relative risk shows that, in comparison with non-smelly modules, smelly modules are 2.785 times more likely to be buggy. This suggests that prioritizing smell-containing modules in bug localization can lead to improved accuracy.

The relative risk of smelly modules varied depending on their smell type. On the one hand, Blob Class, Shotgun Surgery, God Class, Blob Operation, and Intensive Coupling were the smell types with the top five relative risks. The risk of being identified as bugs in modules with these smell types is more than four times higher than modules without these smells. On the other hand, the relative risks for Data Class and Distorted Hierarchy were less than 1, such that the choices of these smells do not necessarily lead to the identification of modules with a high probability of containing bugs.

The three columns on the right in Table 5 present the relative risk of each smell type calculated using only the systems of a specific project group to determine the extent to which the obtained trend is universal. The numbers in parentheses in the table indicate the rank of an item in the project group to which it belongs. As we can see from the table, although several small differences exist, the ranking trend for each project group is similar to the global ranking.

In summary, in comparison with non-smelly modules, smelly modules are 2.785 times more likely to be buggy.

5.7. RQ$_3$: What are the best configurations for smell-aware bug localization as an extension of VSM?

5.7.1. Motivation

As discussed above, when the smell-aware bug localization technique was used in our previous work, we used only one set of configurations, even though many options were available. For example, we can change the granularity of code smells, the aggregator when combining multiple code smells, and the type of code smells. Therefore, when answering this RQ, our goal was to explore the configurations that would yield the best performance.

[4] https://issues.apache.org/jira/browse/HBASE-1795

Table 5
Relative risks of the number of buggy modules included in smelly modules.

	t: Type	# modules	# buggy	Risk_t (%)	Risk_t^* (%)	RR_t	RR_t (Apache)	RR_t (JBoss)	RR_t (Spring)
	Blob Class	1151	246	21.373	2.385	(1) 8.960	(1) 11.401	(2) 6.927	(2) 5.100
	Shotgun Surgery[a]	357	59	16.527	2.411	(2) 6.855	(3) 8.465	(1) 29.963	(4) 4.589
s_5	God Class	6108	959	15.701	2.294	(3) 6.846	(4) 8.407	(3) 6.101	(1) 5.945
	Blob Operation[a]	6802	1021	15.010	2.286	(4) 6.565	(2) 8.539	(5) 3.555	(3) 4.617
	Intensive Coupling[a]	2683	290	10.819	2.384	(5) 4.534	(5) 5.499	(4) 5.199	(6) 3.536
	Data Clumps[a]	8928	702	7.863	2.343	(6) 3.355	(8) 3.921	(6) 2.898	(9) 2.810
s_4	Refused Parent Bequest	1519	119	7.834	2.406	(7) 3.256	(9) 3.782	(9) 1.965	(7) 3.401
	Internal Duplication[a]	3958	306	7.731	2.386	(8) 3.240	(6) 4.387	(7) 2.150	(14) 1.174
	External Duplication[a]	6875	501	7.287	2.367	(9) 3.079	(7) 4.142	(12) 1.226	(11) 1.508
	Feature Envy[a]	9614	669	6.959	2.351	(10) 2.960	(10) 3.479	(10) 1.732	(10) 2.542
s_3	Message Chains[a]	2641	176	6.664	2.401	(11) 2.775	(11) 3.365	(15) 0.000	(5) 4.207
	Schizophrenic Class	3675	222	6.041	2.398	(12) 2.519	(12) 3.005	(13) 0.981	(8) 3.134
s_2	Tradition Breaker	2159	74	3.428	2.415	(13) 1.419	(14) 1.670	(8) 2.056	(12) 1.323
	Sibling Duplication[a]	7860	267	3.397	2.407	(14) 1.411	(13) 1.672	(11) 1.593	(13) 1.232
s_1	Data Class	16028	361	2.252	2.423	(15) 0.930	(15) 0.944	(14) 0.610	(15) 0.903
	Distorted Hierarchy	5	0	0.000	2.419	(16) 0.000	–	(15) 0.000	(16) 0.000
	Total (all smell types)	63953	3668	5.735	2.060	2.785	3.528	2.481	1.969
	All code files	654674	15834	2.419	–	–	–	–	–

[a]Method-level.

Table 6
Accuracy of top 20 configurations.

Rank	g: Granularity	a: Aggregator	s: Type selector	Top 1	Top 5	Top 10	MRR	MAP	# systems
⓪	(Ideal)	(Ideal)	(Ideal)	0.3864	0.6628	0.7595	0.4960	0.3586	257
①	g_3: *both levels*	a_3: *existence*	s_5: *well*	(1) 0.3515	(1) 0.6325	0.7311	(1) 0.4707	(1) 0.3441	201
②	g_3: *both levels*	a_2: *maximum severity*	s_3: *medium rare*	(2) 0.3470	0.6296	(1) 0.7348	(2) 0.4682	(2) 0.3435	229
③	g_3: *both levels*	a_2: *maximum severity*	s_4: *medium*	0.3461	(3) 0.6304	(3) 0.7337	(3) 0.4676	(3) 0.3428	220
④	g_3: *both levels*	a_2: *maximum severity*	s_1: *all*	0.3464	(2) 0.6311	(2) 0.7344	0.4675	0.3427	240
⑤	g_3: *both levels*	a_2: *maximum severity*	s_2: *rare*	0.3451	0.6296	0.7331	0.4671	0.3424	224
⑥	g_3: *both levels*	a_2: *maximum severity*	s_5: *well*	0.3441	0.6263	0.7308	0.4662	0.3419	202
⑦	g_3: *both levels*	a_3: *existence*	s_2: *rare*	(3) 0.3466	0.6301	0.7296	0.4661	0.3411	230
⑧	g_3: *both levels*	a_3: *existence*	s_3: *medium rare*	0.3460	0.6279	0.7302	0.4653	0.3411	230
⑨	g_3: *both levels*	a_3: *existence*	s_4: *medium*	0.3453	0.6286	0.7292	0.4653	0.3410	220
⑩	g_3: *both levels*	a_9: *ave. # of smells*	s_5: *well*	0.3425	0.6231	0.7266	0.4631	0.3404	200
⑪	g_3: *both levels*	a_7: *ave. max severity*	s_5: *well*	0.3411	0.6205	0.7240	0.4622	0.3398	203
⑫	g_3: *both levels*	a_8: *med. max severity*	s_5: *well*	0.3412	0.6207	0.7234	0.4622	0.3397	203
⑬	g_1: *class level*	a_3: *existence*	s_4: *medium*	0.3415	0.6226	0.7228	0.4635	0.3393	167
⑭	g_1: *class level*	a_3: *existence*	s_3: *medium rare*	0.3412	0.6226	0.7236	0.4626	0.3392	185
⑮	g_1: *class level*	a_{10}: *med. # of smells*	s_4: *medium*	0.3409	0.6221	0.7224	0.4632	0.3391	167
⑯	g_1: *class level*	a_9: *ave. # of smells*	s_4: *medium*	0.3409	0.6221	0.7224	0.4632	0.3391	167
⑰	g_1: *class level*	a_9: *ave. # of smells*	s_3: *medium rare*	0.3406	0.6223	0.7230	0.4623	0.3391	185
⑱	g_1: *class level*	a_{10}: *med. # of smells*	s_3: *medium rare*	0.3406	0.6223	0.7230	0.4623	0.3391	185
⑲	g_1: *class level*	a_3: *existence*	s_2: *rare*	0.3406	0.6227	0.7238	0.4623	0.3389	186
⑳	g_1: *class level*	a_{10}: *med. # of smells*	s_5: *well*	0.3393	0.6205	0.7221	0.4620	0.3389	162
74	g_1: *class level*	a_1: *sum of severity*	s_1: *all*	0.3295	0.6177	0.7210	0.4542	0.3345	204

5.7.2. Study design

In our previous study, the technique was limited to textual similarity (*nSim*) and the sum of the severity (*nSev*) as shown when formulating BLI. To answer RQ_3, we utilized the gBLI with three parameters of the code smell configuration c defined in Section 4, as a combination of three granularity levels (g), ten aggregators (a), and five type selectors (s).

We instantiated concrete selections of type selectors. Based on the results we obtained to answer RQ_2, to eliminate smell types that are unlikely to be related with bug-proneness, we created several sets of smell types by excluding those with a lower relative risk and retain only those with a higher relative risk with different boundaries. Finally, we compare the performance of the technique by specifying these five settings.

- s_1: *all smell types* (16): all types of smells,
- s_2: *rare selected smell types* (14): types of smells whose relative risk is greater than 1,
- s_3: *medium rare selected smell types* (12): types of smells whose risk is greater than that of all types of smells (5.735%),
- s_4: *medium selected smell types* (10): types of smells whose relative risk is greater than that of all types of smells (2.785),
- s_5: *well selected smell types* (5): top five types of smells regarding their relative risk; their relative risk is greater than 4.

The concrete types are specified at the left of Table 5.

To compare all configurations, as discussed earlier, we applied our technique to all 150 (= 3 granularity levels × 10 aggregators × 5 type selectors) configurations and calculated the accuracy.

In addition to the five selectors, we also prepared special selectors that use only one smell type to investigate the performance of each smell type. When selecting these selectors together with other perspectives, (1) the smell granularity is automatically assigned to a specific value according to the type, (2) nested aggregators (a_7–a_{10}) are unnecessary because the first aggregation step results in only one value instance, and these aggregators produce the same result as a_2 or a_4, and (3) For class-level smell types, severity-based aggregators (a_1, a_2, a_5, and a_6) and count-based ones (a_3 and a_4) will respectively produce the same result because only one smell instance is assumed to be

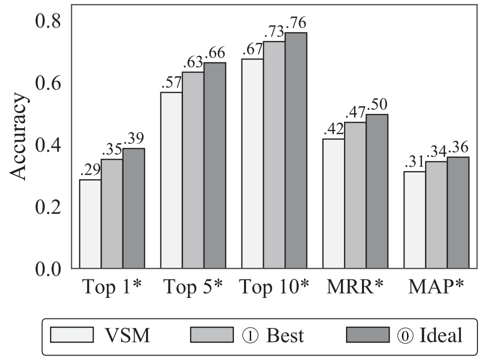


Fig. 4. Accuracy comparison of smell-aware bug localization with the best configuration and VSM.

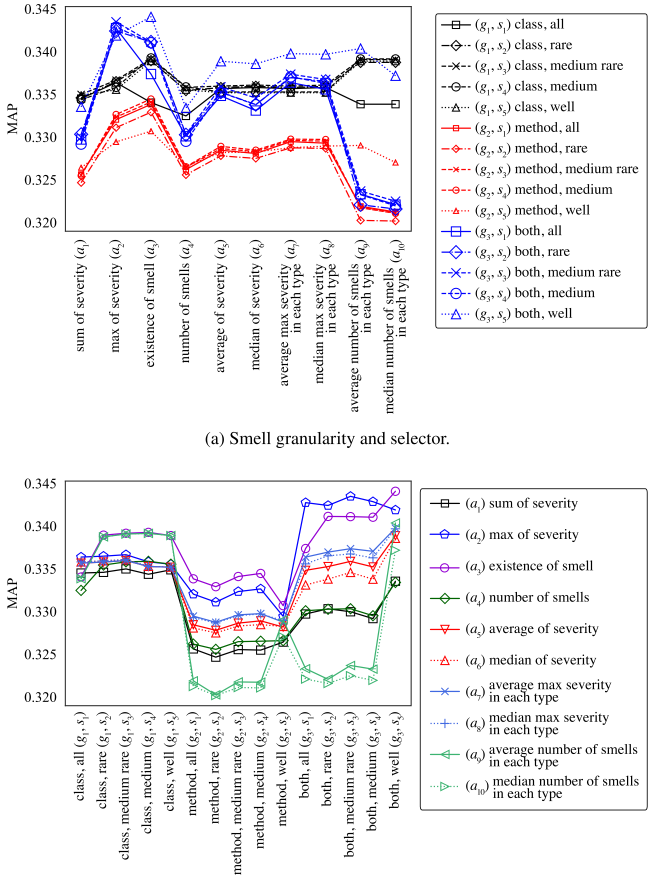
detected, and only the aggregators of a_2 and a_3 are enough to be considered as their representatives. Therefore, we prepared $1 \times 2 \times 7$ (class-level) $+ 1 \times 6 \times 9$ (method-level) $= 68$ configurations. We compare the best configuration of each smell type to determine which smell types contributed to the performance improvements.

5.7.3. Results

The accuracy for the top 20 configurations (① to ⑳) is listed in Table 6, sorted in descending order of MAP. Note that two special configurations were additionally included in the table. (74) is the configuration equivalent to that used in our previous study. ⓪ is a pseudo-ideal configuration that allows a different configuration selection for each system as though we could know the best configuration for each system. The annotated numbers with parentheses highlights represent the rank of each metric value for the top three configurations. In addition, values at the column "# systems" indicate the numbers of systems that the smell-aware approach outperformed the baseline bug localization technique, i.e., the cases where $\alpha > 0$ was used. As we can see, different configurations produce different numbers of systems to succeed, mainly depending on the smell types to be used.

Considering the case of Top 1, Top 5, MRR, and MAP, the best configuration that yields the best performance was the configuration ① ⟨g_3: *both class and method levels*, a_3: *existence of smells*, s_5: *well selected smell types*⟩. On the other hand, when considering Top 10, the combination ② ⟨g_3: *both class and method levels*, a_2: *maximum severity*, s_3: *medium rare selected smell types*⟩ performed the best. In terms of overall performance, ①, ②, and ③ performed well. Based on these observations, providing the technique with an appropriate configuration enables it to significantly outperform the technique developed in our previous study.

Furthermore, to address RQ_1, we used the configuration (74) to conduct the experiment and found the difference in terms of MAP to be statistically insignificant, as shown in Fig. 3. Nevertheless, when we reran the experiment with the configuration ①, we not only observed statistically significant results in all metrics ($p < 0.01$; Cliff's delta: 0.066, 0.065, 0.056, 0.066, and 0.054, all *negligible*), but also an increase in the improvement of all metrics, as shown in Fig. 4. For example, Top 10 increased by 0.056 (from 0.675 to 0.731), which means that the total number of bug reports with gold modules in their top 10-ranked items increased by 391 (from 4676 to 5067). This figure also presents the results obtained using the ideal configuration ⓪, which show additional improvements compared to ①. This result indicates that there remains a scope for improvement when using smell-aware bug localization if we know the best smell configuration per system.



(a) Smell granularity and selector.

(b) Smell aggregator.

Fig. 5. MAP performance of configurations of specific perspectives across the other perspectives.

Analysis on each configuration parameter. We compared the MAP scores according to each configuration parameter to determine the contribution of each parameter to the performance. The following discussion is based on comparisons of the MAP results by fixing all perspectives other than the one to be discussed. Fig. 5 shows the difference in the MAP performance of specific perspectives over the other perspectives. Different combinations of smell granularity (g) and smell selector (s) are compared in Fig. 5(a), whereas the smell aggregators are compared in Fig. 5(b).

The three different colors in Fig. 5(a) specify the smell granularity that was used. The results in the figure show that the configurations using g_1: *class level* and g_3: *both class and method levels* had higher scores depending on the used aggregators, and g_2: *method level* produced worse results in general. In particular, the use of both level smells produced more accurate results when using it with a_2: *maximum severity* or a_3: *existence of smells*. This result suggests that adding method-level smells in addition to class-level smells may be effective in extending the range of smells to be used, but considering all of them may also have a negative effect. For instance, a_1: *sum of severity* and a_4: *number of smells* add up all smells as equivalent, even if class-level smells exist. This kind of smell usage was not effective because method-level smells may decrease the importance of class-level smells. In contrast, a_2: *maximum severity* and a_3: *existence of smells* are considered to be effective because they can consider method-level smells when class-level smells do not exist or when the method-level smells have higher severity than the class-level smells.

Table 7
Accuracy of each configuration when only one smell type is used.

s: Smell type	a: Aggregator	Top 1	Top 5	Top 10	MRR	MAP
God Class	a_3: *existence*	0.3310	0.6148	0.7162	0.4556	0.3349
Blob Operation	a_3: *existence*	0.3311	0.6045	0.7091	0.4499	0.3293
Blob Class	a_2: *severity*	0.3073	0.5914	0.6951	0.4343	0.3218
Feature Envy	a_3: *existence*	0.3086	0.5902	0.6940	0.4328	0.3208
Data Clumps	a_3: *existence*	0.3052	0.5924	0.6963	0.4332	0.3204
Internal Duplication	a_3: *existence*	0.2985	0.5823	0.6884	0.4262	0.3179
Intensive Coupling	a_3: *existence*	0.2966	0.5820	0.6873	0.4253	0.3163
Schizophrenic Class	a_3: *existence*	0.2955	0.5747	0.6811	0.4230	0.3148
External Duplication	a_3: *existence*	0.2919	0.5741	0.6814	0.4212	0.3144
Message Chains	a_3: *existence*	0.2919	0.5754	0.6829	0.4217	0.3140
Refused Parent Bequest	a_3: *existence*	0.2927	0.5726	0.6813	0.4215	0.3139
Shotgun Surgery	a_3: *existence*	0.2899	0.5726	0.6796	0.4203	0.3138
Sibling Duplication	a_3: *existence*	0.2916	0.5719	0.6791	0.4205	0.3134
Data Class	a_3: *existence*	0.2929	0.5728	0.6801	0.4210	0.3131
Tradition Breaker	a_3: *existence*	0.2880	0.5699	0.6768	0.4188	0.3124
Distorted Hierarchy	a_3: *existence*	0.2855	0.5672	0.6747	0.4169	0.3117

In the smell aggregators shown in Fig. 5(b), the configurations using a_2: *maximum severity* and a_3: *existence of smells* produced higher values. Other types of aggregators produced worse results in general, except for the average or median of the number of smells in each type (a_9 and a_{10}) with a certain level of selection in class-level smells (g_1, s_2–s_5). In addition, we found that configurations using median (a_6, a_8, and a_{10}) produced very similar results to those using average (a_5, a_7, and a_9). These results suggest that indicators such as a_2: *maximum severity* and a_3: *existence of smells* should be used.

For the smell selectors, as shown in Fig. 5(b), the use of selection (s_2–s_5) tended to produce more accurate results than the configurations using s_1: *all smell types*. In particular, when using g_3: *both class and method levels*, the use of a_2: *maximum severity* was the best choice at a certain level of selection, whereas the use of a_3: *existence of smells* was more effective if it was used together with s_5: *well selected smell types*. This result suggests the effectiveness of smell selection based on the likelihood of containing bugs, as indicated in Table 5 in general. In addition, the use of the severity degree tends to be more effective if a broader range of smell types are used.

Analysis of individual smell types. The best configurations when their selectors use only one smell type are presented in Table 7. Each row in this table indicates the performance of the best configuration when a specific smell type is used as its selector. Rows are ordered by their MAP score. Although several smell types, such as God Class, Blob Operation, or Blob Class, outperformed other smell types, no one outperformed the best configurations in Table 6. This result shows that an awareness of multiple smell types improves the performance to a greater extent than only one specific smell type.

Distribution of AP improvements for each bug report. To improve our understanding of the performance improvement, we analyzed the distribution of AP improvements of 6931 bug reports when applying smell-aware bug localization to the VSM results. The distribution is visualized in Fig. 6. In this figure, the black line is plotted to express the AP deltas ($\Delta AP = AP_{\text{smell-aware}} - AP_{\text{VSM}}$) in descending order, obtained using the best configuration ①. As is shown, the smell-aware bug localization improved the overall accuracy because the total improvements (top left) exceeded the total decreases (bottom right). However, the use of the smell-aware bug localization did not improve the values of all bug reports; instead, certain values were less accurate. In this result, out of 6931 bug reports, 1809 of AP increased, whereas 2499 decreased. However, in most of them, the value of delta was small, and for those with an absolute delta greater than 0.01, 1543 increased, whereas 1138 decreased. In particular, 803 reports improved and 265 did not improve when the absolute delta was greater than 0.1. We consider that the accuracy to have improved because the number of bug reports with large improvements was relatively more extensive than that with large decreases. The figure also includes plots of the original configuration (74) and the second-best configuration ②. Clearly, the degree of improvement by the best configuration is much higher than that of the original configuration. Moreover, the difference with the second-best configuration is small.

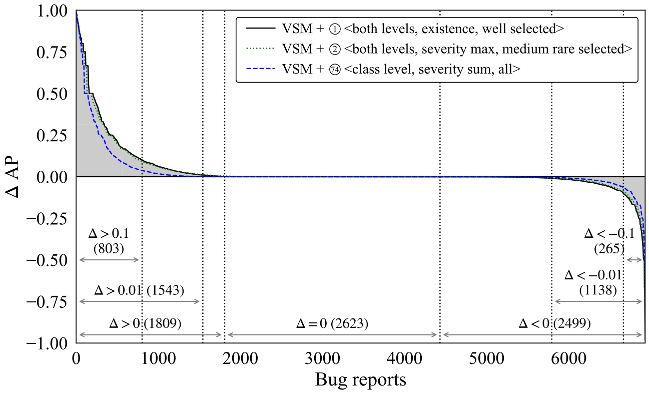
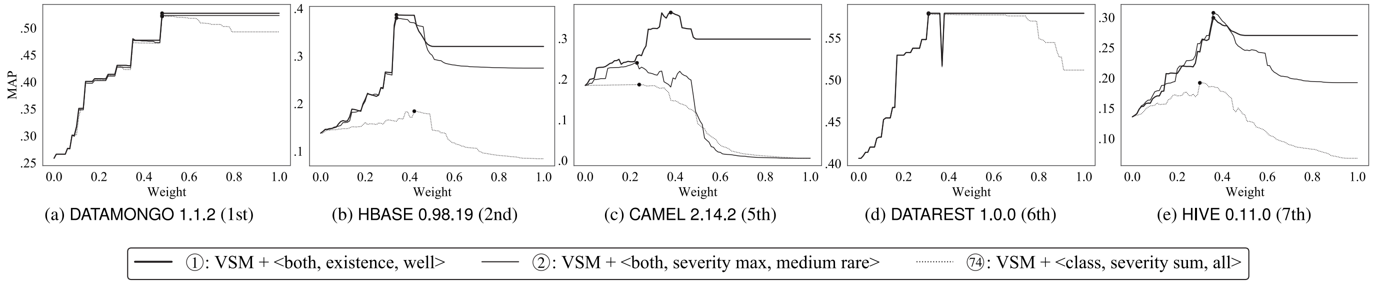


Fig. 6. Distribution of AP improvements.

Benefited and non-benefited systems. Fig. 7 shows the results of the top systems with the biggest improvement in MAP when using the best configuration ①. We picked up first, second, fifth, sixth, and seventh-top systems because they are the top systems when picking up only one version from the same project. In the figure, each graph shows the MAP value at different α for each system. For comparison, the results of the two best configurations as well as the original configuration, i.e., ①, ②, and (74), are plotted. The point for $\alpha = 0$ refers to the accuracy when using only the baseline IR-based bug localization technique, i.e., VSM, whereas $\alpha = 1$ refers to the accuracy when using only the code smell property to localize bugs. We can see two typical shapes in these plots:

- *Mountain-shaped*: A certain level of blending of IR-based bug localization and the smell information gave the best result, i.e., $0 \notin A \wedge 1 \notin A$, where A is the set of best α values for the setting. Including HBASE 0.98.19, CAMEL 2.14.2, and HIVE 0.11.0, the plots of 196 and 225 out of 309 systems in ① and ② follow this style, respectively.
- *Plateau-shaped*: When α exceeds a certain value, MAP stayed at the highest value, i.e., $0 \notin A \wedge 1 \in A$. This case shows that smells caught the buggy location without the help of the textual similarity of the IR-based bug localization. Including



(a) DATAMONGO 1.1.2 (1st) (b) HBASE 0.98.19 (2nd) (c) CAMEL 2.14.2 (5th) (d) DATAREST 1.0.0 (6th) (e) HIVE 0.11.0 (7th)

Fig. 7. Distribution of MAP values according to α parameter.

1.1.2 and DATAREST 1.0.0, the plots of 5 and 4 out of 309 systems in ① and ② follow this style, respectively.

Also, we can see that the best configuration did not always lead to the best result; for example, in Fig. 7(e), the second-best configuration led to the best result. Note that 101 of 309 systems used the parameter of $\alpha = 0$ when using the best configuration ①. The most typical case for this situation is simply that the buggy portions to be fixed were not smelly.

In conclusion, the configurations using *both class and method levels* for the granularity yielded the best result. In terms of the combinations of the aggregator and the selector, the *existence* of *well selected types of smells* or *maximum severity* of *medium rare selected types of smells* yielded the best results.

5.8. RQ_4: Is the performance of smell-aware bug localization superior to that of state-of-the-art bug localization techniques?

5.8.1. Motivation

When addressing RQ_1 and RQ_3, we found that the smell-aware bug localization technique can improve the performance when combined with the VSM technique. However, many bug localization techniques have been proposed to improve the VSM technique such as rVSM, BugLocator, BRTracer, BLUiR, and AmaLgam. The goal of this study is to verify whether the smell-aware bug localization technique can also be used to improve bug localization techniques other than VSM.

5.8.2. Study design

We implemented the smell-aware bug localization technique using six existing bug localization techniques as baselines. Specifically, we used the output score of each technique as *nScore* in Section 4. For *nSmell*, we apply the best configurations discussed in RQ_3, that is, ① and ② in Table 6.

Finally, we compare the accuracy of the ranking produced using gBLI and each baseline technique.

5.8.3. Results

The results of each technique using with ① $\langle g_3$: *both class and method levels*, a_3: *existence of smells*, s_5: *well selected smell types*$\rangle$ as the smell configuration are shown in Fig. 8. Although we applied two configurations (① and ②), we reached almost the same conclusion for each configuration. To save space, we mainly selected ① to explain the details and clarified major differences if exist.

Fig. 8(a) shows the result of the rVSM technique. The results obtained with the smell-aware bug localization technique are an improvement relative to the baseline by approximately 11.0%, 5.4%, 3.7%, 6.4%, and 4.7% in relative comparison (0.043, 0.036, 0.028, 0.033, and 0.018 in absolute comparison), for Top 1, Top 5, Top 10, MRR, and MAP, respectively. All of these improvements are statistically significant (Cliff's delta: 0.043, 0.036, 0.028, 0.041, and 0.030, all *negligible*). This indicates that smell information is useful even with a bug localization technique that uses information about the size of the source code. This result suggests that the smell-aware bug localization technique is effective not only because of the size of the source code but also because of other factors.

In the case of BugLocator, as shown in Fig. 8(b), a similar result was observed. All of the improvements except for MAP are statistically significant: 8.6%, 4.3%, 3.2%, 4.8%, and 3.1% in relative comparison (0.036, 0.030, 0.025, 0.026, and 0.013 in absolute comparison) for Top 1, Top 5, Top 10, MRR, and MAP (Cliff's delta: 0.036, 0.030, 0.025, 0.033, and 0.022, all *negligible*), respectively. Note that the MAP difference was statistically significant when using the configuration ②. This result suggests that our technique even improves the technique that used information about bug reports in the past.

However, in the cases of BRTracer, BLUiR, and AmaLgam in Figs. 8(c), 8(d), and 8(e), statistically significant improvements in all the metrics except MAP are obvious. Specifically, the improvement in MAP in BRTracer is only 1.7%, which is the lowest improvement among all the projects.

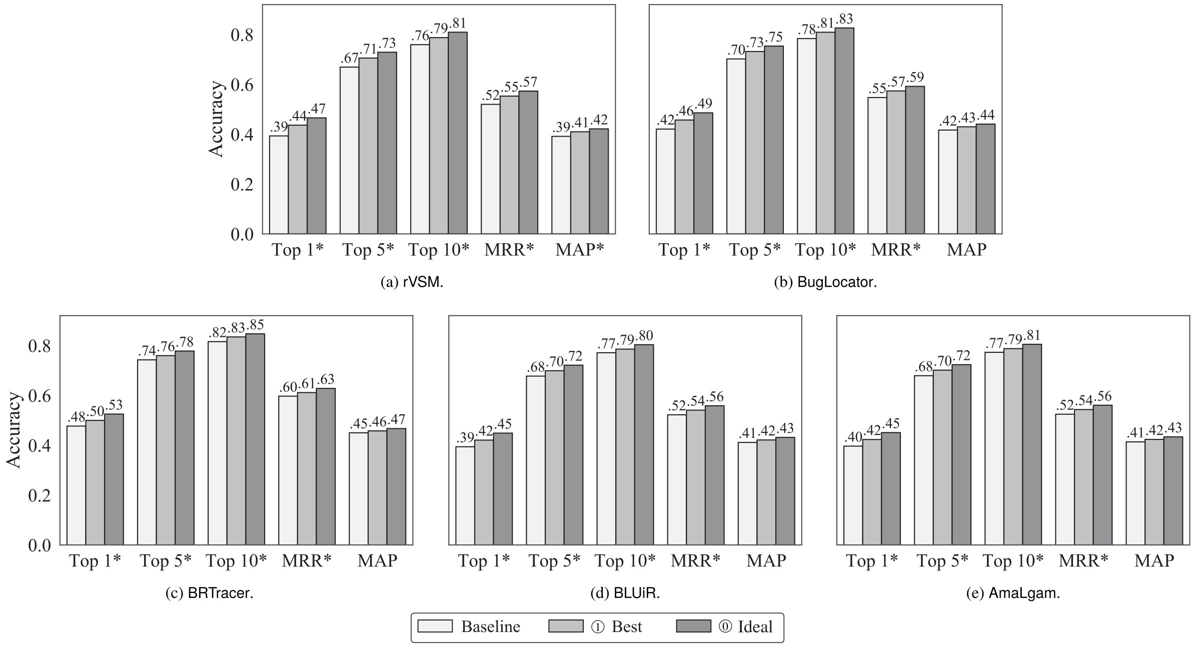
In conclusion, optimization of the configuration of the smell-aware bug localization technique can improve state-of-the-art bug localization techniques.

6. Threats to validity

6.1. Internal validity

In this study, the weight α was assigned the optimal value to maximize each accuracy metric of each version. This is intended to avoid the possibility of not being able to observe the effect of smells because of the effect on the accuracy as a result of the choice of α. However, in RQ_1 and RQ_4, we also discussed the extent to which the smell-aware bug localization technique is superior to the baseline. Therefore, the threat that remains is that the used optimal α values differ from the practical values of α. In particular, the possibility of such cases occurring in versions with a small number of bug reports is high because the values of α might be biased toward those bug reports. Therefore, we conducted our experiments by excluding versions with fewer than five bug reports. This exclusion prevented the value of α from being over-optimized in a version with fewer bug reports.

Another threat is the accuracy of the output of Bench4BL, which we used in our study. As mentioned in Section 5.3.4, we mitigated this threat by excluding inconsistent results from the output of Bench4BL. However, the validity of the output depends on the quality of Bench4BL, even for the consistent results. Although we excluded two bug localization techniques in Bench4BL as shown in Section 5.3.4, we have not verified the correctness of the remaining four techniques (and the two derived from them). The possibility of incorrectness in implementing these bug localization techniques still exists. However, we believe that these



(a) rVSM.

(b) BugLocator.

(c) BRTracer.

(d) BLUiR.

(e) AmaLgam.

Fig. 8. Accuracy of the smell-aware bug localization technique relative to that of the baseline.

four implementations have a certain degree of correctness. As shown in Fig. 2, there is a relationship between the techniques in terms of the use of additional information, and it is expected that the more additional information is used, the higher the accuracy becomes. The accuracy attained in this study did not contradict this relationship, suggesting that the implementations may yield the expected accuracy.

Moreover, it should be noted that Bench4BL can only run on file-level bug localization. Wang et al. (2015) suggested that the results of most bug localization techniques at the file level still leave developers with a large amount of code to examine. Therefore, it might be beneficial to conduct the same experiment at the class or method level on different datasets. Noteworthy is that, although our previous work was conducted on method-level modules, a large-scale method-level benchmark dataset is not yet available. We continue the discussion on method-level bug localization in Section 7.4.

Finally, the accuracy of smell detection in inFusion may be a threat. Manually validating the smells detected in inFusion to exclude the presence of false positives and negatives remains a future task. We continue the discussion on the false positives in Section 7.2.

6.2. External validity

Although we mitigated the threat of external validity by using a sufficient number of bug reports, we limited our attention to Java systems. Moreover, we only used open-source systems. Therefore, performing similar studies on industrial systems may be beneficial. In addition, although we used the largest available bug localization dataset in this study, the optimal configuration presented in this paper might have different results on other datasets. Moreover, we considered 16 types of code smells in this study, yet other types of code smells or other methods to calculate the smell severity (Taba et al., 2013) that were not considered in this study are also available. In addition, we only used inFusion as the smell detector, despite the existence of other possible smell detectors that were not considered in this study. As a result, they may have a different effect on bug localization. Finally, because our experiments on bug localization were only conducted at the file level owing to the limitations of Bench4BL, we would need to conduct experiments at other levels, e.g., the method level.

6.2.1. Conclusion validity

Although we performed statistical tests (Wilcoxon signed-rank tests) and confirmed statistical significance between bug localization techniques, the obtained effect size computed via Cliff's delta was very small. This indicates a possibility that the significance came from a large sample set in our experiment, and the essential effect might be negligible. We will discuss this point in Section 7.1.

7. Discussion

7.1. Accuracy of bug localization techniques

Even though our results improved significantly by using information about the code smells in combination with the bug localization technique, one might consider the improvements to be small, i.e., improvements of 10.4–23.0% in relative comparison. Actually, the obtained effect size computed via Cliff's delta was all little. However, we argue that these improvements are meaningful in the context of bug localization. As shown in Fig. 8, a comparison of the accuracy of each baseline technique reveals that the improvements are on the same scale. For example, even though BugLocator was improved relative to rVSM, MAP improved by only 6.5% in relative comparison (from 0.391 to 0.417; 0.025 in absolute comparison). This means that improving the bug localization accuracy is generally difficult, even if additional information sources, such as past bug reports or history information, were to be utilized. Smell-aware bug localization can significantly improve the accuracy of state-of-the-art bug localization techniques in similar amounts. In addition, it is noteworthy that the smell-aware bug localization technique uses only the source code and does not require additional information. Therefore, we suggest that source code characteristics, such as code smells, should be considered when performing bug localization.

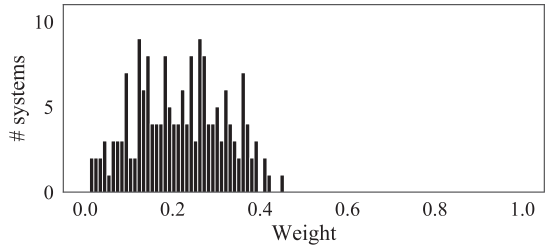


Fig. 9. Distribution of the optimal α values.

7.2. Accuracy of detected smells

In some cases, smell detectors might produce false positives. For example, a parser class having God Class is not considered problematic because its scope is generally large, and refactoring it might even reduce the comprehensibility of the class (Fontana et al., 2016, 2015a). Several existing studies have reported the accuracy of smells detected by inFusion (Fernandes et al., 2016; Paiva et al., 2017; Fontana et al., 2012), but the reported accuracy varies widely and covers only a small subset of smell types.

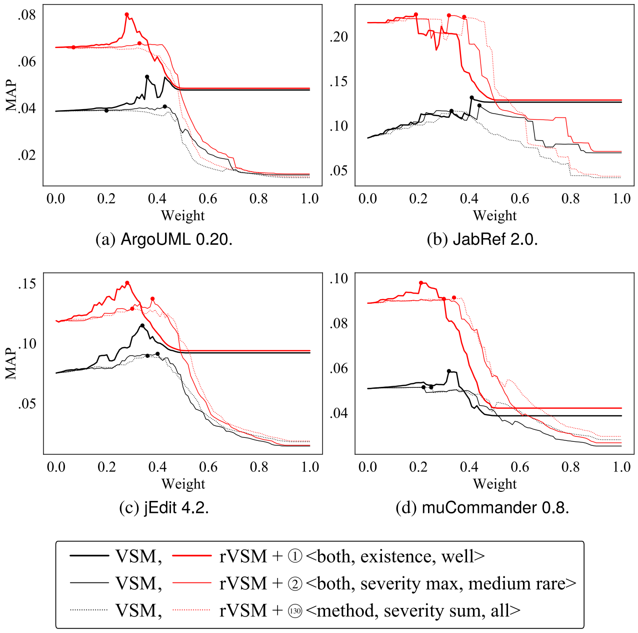
To mitigate this threat, the authors manually verified sampled smell instances detected by inFusion. We followed the false positive catalog by Fontana et al. (2016, 2015a). For the seven smell types listed in the catalog with their false positive detection strategy (Blob Class, Data Class, God Class, Blob Operation, Feature Envy, Message Chains, and Shotgun Surgery), we randomly selected five smell instances for each type, and we collected 35 smell instances in total. We selected one instance from each of the top five projects with the highest number of smell instances in total, resulting in five instances for each type. For each instance, two of the authors independently judged whether it met the condition to be regarded as a false positive according to the false positive detection strategy in the catalog. In the case of two authors' decisions being in conflict, we conducted a discussion to reach a consensus. As a result, four false positives out of 35 were identified, yielding a precision of 0.89. Although the number of extracted samples was very small, this result suggests that a certain percentage of smells used in our study were correct instances to be regarded as smells. We conclude that false positives may have little effect on the results of this study.

However, because the process of identifying false positives was performed by the authors, who are not the main developers of the projects used for this study, we cannot ensure the completeness of the identified results. Furthermore, our sampling approach cannot confirm the recall of the detected smells. Both are still regarded as a threat to validity.

7.3. Distribution of α values

We studied the distribution of the optimal value of α for each system. We investigated 309 systems using MAP as the evaluation metric when the optimal configuration obtained from RQ_3, i.e., ① ⟨g_3: *both class and method levels*, a_3: *existence of smells*, s_5: *well selected smell types*⟩, was used. Note that we observed multiple α values that can maximize the MAP value in 85 out of 309 systems, such as DATAMONGO 1.1.2 as shown in Fig. 7. For the sake of simplicity of the analysis, we excluded these systems and selected the other 224 systems as the target of the subsequent investigation.

The systems were broadly divided into two categories: those who could utilize smell-based scores effectively and those for which the detected smells did not work at all. On the one hand, in 49 of 224 systems, the setting of $\alpha = 0$ produced their optimal ranking. This means that any blending of smell-based scores reduced the accuracy of the resulting rankings; i.e., the use of smells did not improve the accuracy of these systems. On the other hand, in the remaining 175 systems, blending the smell-based score improved the ranking compared to the base IR-based bug localization using VSM. Fig. 9 shows a histogram representing the distribution of optimal α values of the 175 systems. The average value of the obtained optimal α values was 0.215, and all the values were less than 0.5. For these systems to which the smell information contributed positively, a weak blending of the smell-based score with the IR-based score tends to improve the ranking well. Although this analysis is limited because of the selected instances of α values to be used, we think that this average value can be regarded as a representative. The prediction of an appropriate α value, for example, using a machine learning technique, contributes to bug localization improvement. For example, it may be possible to compute an appropriate α value for a project version by using the results obtained from past versions of the project. Note that such an approach should deal with the size variation of the versions and some inappropriate versions that smell-aware bug localization is not suitable, as we studied in this section.



(a) ArgoUML 0.20.

(b) JabRef 2.0.

(c) jEdit 4.2.

(d) muCommander 0.8.

Fig. 10. Distribution of MAP values in method-level bug localization.

7.4. Application to method-level bug localization

In our previous paper (Takahashi et al., 2018), we have studied not only class-level but also method-level bug localization. However, the study in this paper focused on the class level. This is because Bench4BL is based on files, which does not provide bug localization at the method level, and no bug localization benchmarking framework supports the application of bug localization techniques at the method level.

To preliminary confirm whether the best configurations obtained in the study are effective for method-level bug localization, we manually adapted four systems used in the previous paper to the Bench4BL framework and obtained method-level application results. We obtained the method level matching results by converting source code snapshots to those at the method level using FinerGit (Higo et al., 2020). The results are shown in

Fig. 10. Similarly to Fig. 7, the plots in the figure show the results using the two best configurations (① and ②) and the original configuration at the method level ((130)) using VSM and rVSM as the baseline IR techniques. As we can see from the plots, the use of the best configurations improved the MAP for all systems in both VSM and rVSM. Although a detailed study on the method level bug localization is subject to future work, these results suggest the applicability of the proposed smell-aware bug localization technique to the method level.

8. Conclusion

In this study, we replicated the work conducted with our previous smell-aware bug localization technique on a large-scale dataset and confirmed significant performance improvement. We proposed a generalized smell-aware bug localization technique to derive the optimal configurations for code smell information. We found that the optimal configuration entails the use of granularity that reflects both class- and method-level smells and the maximum severity when aggregating a certainly selected types of code smells or the existence of very limited types of smells. Finally, we combined our proposed technique with different baseline techniques and found that the performance improved significantly. These results suggest that code smells can be used to effectively improve existing bug localization techniques without the need for additional information.

Code smell detection does not require more inputs than bug localization in general. Although the improvement was slight, the application of the smell-aware approach can improve bug localization, which is applicable in many situations. Our study also revealed that there are situations where the effect of the smell-aware approach was negative. It is desirable to develop further techniques that use smell information more effectively to avoid losing the accuracy of the baseline bug localization technique. The use of machine learning techniques or other data fusion techniques might be effective rather than a simple linear combination that we used in this paper.

In the future, we also aim to specify the value of α for each version of a specific project. For example, we can set the value of α based on the optimal value of previous versions or by using a machine learning approach.

An appendix including the experimental materials is available in online (Takahashi et al., 2021).

CRediT authorship contribution statement

Aoi Takahashi: Conceptualization, Methodology, Software, Validation, Investigation, Writing - original draft, Writing - review & editing. **Natthawute Sae-Lim:** Conceptualization, Methodology, Validation, Writing - original draft, Writing - review & editing. **Shinpei Hayashi:** Conceptualization, Methodology, Software, Validation, Investigation, Writing - original draft, Writing - review & editing, Project administration. **Motoshi Saeki:** Methodology, Validation, Writing - original draft, Writing - review & editing.

Declaration of competing interest

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

Acknowledgments

This work was partly supported by JSPS KAKENHI Grant Number JP18K11238.

References

Blei, D.M., Ng, A.Y., Jordan, M.I., 2003. Latent dirichlet allocation. J. Mach. Learn. Res. 3 (Jan), 993–1022.

Brown, W.H., Malveau, R.C., McCormick, H.W., Mowbray, T.J., 1998. AntiPatterns: Refactoring Software, Architectures, and Projects in Crisis. John Wiley & Sons, Inc.

Chaparro, O., Florez, J.M., Marcus, A., 2017. Using observed behavior to reformulate queries during text retrieval-based bug localization. In: Proceedings of the 33rd IEEE International Conference on Software Maintenance and Evolution, ICSME'17. pp. 376–387.

Chaparro, O., Florez, J.M., Marcus, A., 2019. Using bug descriptions to reformulate queries during text-retrieval-based bug localization. Empir. Softw. Eng. 24 (5), 2947–3007.

Cormack, G.V., Lynam, T.R., 2006. Statistical precision of information retrieval evaluation. In: Proceedings of the 29th International ACM SIGIR Conference on Research and Development in Information Retrieval, SIGIR'06. pp. 533–540.

Craswell, N., 2009. Mean reciprocal rank. In: Encyclopedia of Database Systems. Springer, p. 1703.

Dit, B., Moritz, E., Poshyvanyk, D., 2012. A TraceLab-based solution for creating, conducting, and sharing feature location experiments. In: Proceedings of the 20th IEEE International Conference on Program Comprehension, ICPC'12. pp. 203–208.

Dit, B., Revelle, M., Gethers, M., Poshyvanyk, D., 2013. Feature location in source code: A taxonomy and survey. J. Softw.: Evol. Process 25 (1), 53–95.

Fernandes, E., Oliveira, J., Vale, G., Paiva, T., Figueiredo, E., 2016. A review-based comparative study of bad smell detection tools. In: Proceedings of the 20th International Conference on Evaluation and Assessment in Software Engineering, No. 18. EASE'16. pp. 1–12.

Fontana, F.A., Braione, P., Zanoni, M., 2012. Automatic detection of bad smells in code: An experimental assessment. J. Object Technol. 11 (2), 5:1–38.

Fontana, F.A., Dietrich, J., Walter, B., Yamashita, A., Zanoni, M., 2015a. Preliminary Catalogue of Anti-Pattern and Code Smell False Positives. Tech. Rep. RA-5/15, Poznan University of Technology, pp. 1–28.

Fontana, F.A., Dietrich, J., Walter, B., Yamashita, A., Zanoni, M., 2016. Antipattern and code smell false positives: Preliminary conceptualization and classification. In: Proceedings of the 23rd IEEE International Conference on Software Analysis, Evolution, and Reengineering, SANER'16. pp. 609–613.

Fontana, F.A., Ferme, V., Zanoni, M., Roveda, R., 2015b. Towards a prioritization of code debt: A code smell intensity index. In: Proceedings of the 7th IEEE International Workshop on Managing Technical Debt, MTD'15. pp. 16–24.

Fowler, M., 1999. Refactoring: Improving the Design of Existing Code. Addison-Wesley Professional.

Ganea, G., Verebi, I., Marinescu, R., 2017. Continuous quality assessment with inCode. Sci. Comput. Program. 134, 19–36.

Gay, G., Haiduc, S., Marcus, A., Menzies, T., 2009. On the use of relevance feedback in IR-based concept location. In: Proceedings of the 25th IEEE International Conference on Software Maintenance, ICSM'09. pp. 351–360.

Gethers, M., Dit, B., Kagdi, H., Poshyvanyk, D., 2012. Integrated impact analysis for managing software changes. In: Proceedings of the 34th IEEE International Conference on Software Engineering, ICSE'12. pp. 430–440.

Guerrouj, L., Kermansaravi, Z., Arnaoudova, V., Fung, B.C., Khomh, F., Antoniol, G., Guéhéneuc, Y.-G., 2017. Investigating the relation between lexical smells and change-and fault-proneness: an empirical study. Softw. Qual. J. 25 (3), 641–670.

Higo, Y., Hayashi, S., Kusumoto, S., 2020. On tracking Java methods with Git mechanisms. J. Syst. Softw. 165 (110571), 1–13.

Hunt, A., Thomas, D., 2000. The Pragmatic Programmer: From Journeyman to Master. Addison-Wesley.

Khomh, F., Di Penta, M., Guéhéneuc, Y.-G., Antoniol, G., 2012. An exploratory study of the impact of antipatterns on class change-and fault-proneness. Empir. Softw. Eng. 17 (3), 243–275.

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

Landauer, T.K., Foltz, P.W., Laham, D., 1998. An introduction to latent semantic analysis. Discourse Process. 25 (2–3), 259–284.

Lanza, M., Marinescu, R., 2007. Object-Oriented Metrics in Practice. Springer Science & Business Media.

Le, T.-D.B., Thung, F., Lo, D., 2017. Will this localization tool be effective for this bug? Mitigating the impact of unreliability of information retrieval based bug localization tools. Empir. Softw. Eng. 22 (4), 2237–2279.

Lee, J., Kim, D., Bissyandé, T.F., Jung, W., Le Traon, Y., 2018. Bench4BL: Reproducibility study on the performance of IR-based bug localization. In: Proceedings of the 27th ACM International Symposium on Software Testing and Analysis, ISSTA'18. pp. 61–72.

Lukins, S.K., Kraft, N.A., Etzkorn, L.H., 2010. Bug localization using latent dirichlet allocation. Inf. Softw. Technol. 52 (9), 972–990.

Marinescu, R., 2012. Assessing technical debt by identifying design flaws in software systems. IBM J. Res. Dev. 56 (5), 9:1–13.

Marinescu, C., Marinescu, R., Mihancea, P., Ratiu, D., Wettel, R., 2005. iPlasma: An integrated platform for quality assessment of object-oriented design. In: Proceedings of the 21st IEEE International Conference on Software Maintenance (ICSM'05) Industrial and Tool Volume. pp. 77–80.

Martin, R.C., 2007. Agile Software Development: Principles, Patterns, and Practices. Prentice Hall.

Moha, N., Guéhéneuc, Y.-G., Duchien, L., Meur, A.-F.L., 2010. DECOR: A method for the specification and detection of code and design smells. IEEE Trans. Softw. Eng. 36 (1), 20–36.

Moreno, L., Bavota, G., Haiduc, S., Di Penta, M., Oliveto, R., Russo, B., Marcus, A., 2015. Query-based configuration of text retrieval solutions for software engineering tasks. In: Proceedings of the 10th Joint Meeting on Foundations of Software Engineering, ESEC/FSE'15. pp. 567–578.

Moreno, L., Treadway, J.J., Marcus, A., Shen, W., 2014. On the use of stack traces to improve text retrieval-based bug localization. In: Proceedings of the 30th IEEE International Conference on Software Maintenance and Evolution, ICSME'14. pp. 151–160.

Nguyen, A.T., Nguyen, T.T., Al-Kofahi, J., Nguyen, H.V., Nguyen, T.N., 2011. A topic-based approach for narrowing the search space of buggy files from a bug report. In: Proceedings of the 26th IEEE/ACM International Conference on Automated Software Engineering, ASE'11. pp. 263–272.

Paiva, T., Damasceno, A., Figueiredo, E., Sant'Anna, C., 2017. On the evaluation of code smells and detection tools. J. Softw. Eng. Res. Dev. 5 (7), 1–28.

Palomba, F., Zanoni, M., Fontana, F.A., De Lucia, A., Oliveto, R., 2017. Toward a smell-aware bug prediction model. IEEE Trans. Softw. Eng. 45 (2), 194–218.

Rao, S., Kak, A., 2011. Retrieval from software libraries for bug localization: A comparative study of generic and composite text models. In: Proceedings of the 8th ACM Working Conference on Mining Software Repositories, MSR'11. pp. 43–52.

Riel, A.J., 1996. Object-Oriented Design Heuristics. Addison-Wesley.

Romano, J., Kromrey, J.D., Coraggio, J., Skowronek, J., 2006. Appropriate statistics for ordinal level data: Should we really be using t-test and Cohen's d for evaluating group differences on the NSSE and other surveys? In: Proceedings of the 2006 Annual Meeting of the Florida Association of Institutional Research, FAIR'06, pp. 1–33.

Saha, R.K., Lease, M., Khurshid, S., Perry, D.E., 2013. Improving bug localization using structured information retrieval. In: Proceedings of the 28th IEEE International Conference on Automated Software Engineering, ASE'13. pp. 345–355.

Salton, G., Wong, A., Yang, C.-S., 1975. A vector space model for automatic indexing. Commun. ACM 18 (11), 613–620.

Shi, Z., Keung, J., Bennin, K.E., Zhang, X., 2018. Comparing learning to rank techniques in hybrid bug localization. Appl. Soft Comput. 62, 636–648.

Sistrom, C.L., Garvan, C.W., 2004. Proportions, odds, and risk. Radiology 230 (1), 12–19.

Stefano, M.D., Gambardella, M.S., Pecorelli, F., Palomba, F., Lucia, A.D., 2020. cASpER: A plug-in for automated code smell detection and refactoring. In: Proceedings of the ACM International Conference on Advanced Visual Interfaces, No. 79. AVI'20. pp. 1–3.

Taba, S.E.S., Khomh, F., Zou, Y., Hassan, A.E., Nagappan, M., 2013. Predicting bugs using antipatterns. In: Proceedings of the 35th IEEE International Conference on Software Maintenance, ICSM'13. pp. 270–279.

Takahashi, A., Sae-Lim, N., Hayashi, S., Saeki, M., 2018. A preliminary study on using code smells to improve bug localization. In: Proceedings of the 26th IEEE International Conference on Program Comprehension, ICPC'18. pp. 324–327.

Takahashi, A., Sae-Lim, N., Hayashi, S., Saeki, M., 2021. Online Appendix of "An Extensive Study on Smell-Aware Bug Localization". Zenodo, http://dx.doi.org/10.5281/zenodo.4681242.

Tantithamthavorn, C., Ihara, A., Matsumoto, K., 2013. Using co-change histories to improve bug localization performance. In: Proceedings of the 14th IEEE International Conference on Software Engineering, Artificial Intelligence, Networking and Parallel/Distributed Computing, SNPD'13. pp. 543–548.

Tsantalis, N., Chaikalis, T., Chatzigeorgiou, A., 2008. JDeodorant: Identification and removal of type-checking bad smells. In: Proceedings of the 12th European Conference on Software Maintenance and Reengineering, CSMR'08. pp. 329–331.

Wang, S., Lo, D., 2014. Version history, similar report, and structure: Putting them together for improved bug localization. In: Proceedings of the 22nd International Conference on Program Comprehension, ICPC'14. pp. 53–63.

Wang, Q., Parnin, C., Orso, A., 2015. Evaluating the usefulness of IR-based fault localization techniques. In: Proceedings of the 24th ACM International Symposium on Software Testing and Analysis, ISSTA'15. pp. 1–11.

Wen, M., Wu, R., Cheung, S.-C., 2016. Locus: Locating bugs from software changes. In: Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering, ASE'16. pp. 262–273.

Wilcoxon, F., Katti, S., Wilcox, R.A., 1970. Critical values and probability levels for the Wilcoxon rank sum test and the Wilcoxon signed rank test. Sel. Tables Math. Stat. 1, 171–259.

Wong, C.-P., Xiong, Y., Zhang, H., Hao, D., Zhang, L., Mei, H., 2014. Boosting bug-report-oriented fault localization with segmentation and stack-trace analysis. In: Proceedings of the 30th IEEE International Conference on Software Maintenance and Evolution, ICSME'14. pp. 181–190.

Yamashita, A., Moonen, L., 2012. Do code smells reflect important maintainability aspects? In: Proceedings of the 28th IEEE International Conference on Software Maintenance, ICSM'12. pp. 306–315.

Yamashita, A., Moonen, L., 2013. Exploring the impact of inter-smell relations on software maintainability: An empirical study. In: Proceedings of the 35th IEEE International Conference on Software Engineering, ICSE'13. pp. 682–691.

Youm, K.C., Ahn, J., Lee, E., 2017. Improved bug localization based on code change histories and bug reports. Inf. Softw. Technol. 82, 177–192.

Zhou, J., Zhang, H., Lo, D., 2012. Where should the bugs be fixed? More accurate information retrieval-based bug localization based on bug reports. In: Proceedings of the 34th IEEE International Conference on Software Engineering, ICSE'12. pp. 14–24.