Garnier *et al. Journal of Software Engineering Research and Development* (2017) 5:6
DOI 10.1186/s40411-017-0040-2

Journal of Software Engineering Research and Development

RESEARCH Open Access

On the influence of program constructs on bug localization effectiveness

A study of 20 C# projects

Marcelo Garnier*, Isabella Ferreira and Alessandro Garcia

*Correspondence:
mgarnier@inf.puc-rio.br
OPUS Research Group, Informatics Department, PUC-Rio, Rua Marquês de São Vicente, 225, Rio de Janeiro, Brazil

Abstract

Software projects often reach hundreds or thousands of files. Therefore, manually searching for code elements that should be changed to fix a failure is a difficult task. Static bug localization techniques provide cost-effective means of finding files related to the failure described in a bug report. Structured information retrieval (IR) has been successfully applied by techniques such as BLUiR, BLUiR+, and AmaLgam. However, there are significant shortcomings on how these techniques were evaluated. First, virtually all evaluations have been limited to very few projects written in only one object-oriented programming language, particularly Java. Second, it might be that particular constructs of different programming languages, such as C#, play a role on the effectiveness of bug localization techniques. However, little is known about this phenomenon. Third, the experimental setup for most of the bug localization studies make simplistic assumptions that do not hold on real-world scenarios, thereby raising doubts about the reported effectiveness of existing techniques. In this article, we evaluate BLUiR, BLUiR+, and AmaLgam on 20 C# projects, addressing the aforementioned shortcomings from previous studies. Then, we extend AmaLgam's algorithm to understand if structured information retrieval can benefit from the use of a wider range of program constructs, including C# constructs inexistent in Java. We also perform an analysis of the influence of program constructs to bug localization effectiveness using Principal Component Analysis (PCA). Our analysis points to *Methods* and *Classes* as the constructs that contribute the most to the effectiveness of bug localization. It also reveals a significant contribution from *Properties* and *String literals*, constructs not considered in previous studies. Finally, we evaluate the effects of changing the emphasis on particular constructs by making another extension to AmaLgam's algorithm, enabling the specification of different weights for each construct. Our results show that fine-tuning these weights may increase the effectiveness of bug localization in projects structured with a specific programming language, such as C#.

Keywords: Bug localization, Structured information retrieval, Bug reports

1 Introduction

Software defects (bugs) are a serious concern for developers and maintainers. It is widely known that the later a failure is detected, higher the cost to fix it. To fix the bug that caused a failure, one must first know *where* the bug is located. The activity of finding the defective source code elements that led to a failure is called *bug localization* (Lukins et al. 2010). Effective methods for automatically locating bugs from bug reports are highly desirable (Saha et al. 2013), as they would shorten bug-fixing time, consequently reducing software maintenance costs (Zhou et al. 2012). Given the ever-increasing size and complexity of software projects, manual bug localization is a lengthy and challenging task, which motivates research for automated bug localization techniques (Rahman et al. 2011; Saha et al. 2013; Sisman and Kak 2012; Wang and Lo 2014; Zhou et al. 2012).

To foster the process of effectively identifying source code that is relevant to a particular bug report, a number of techniques have been developed using information retrieval (IR) models such as Latent Dirichlet Allocation (LDA) (Lukins et al. 2010), Latent Semantic Analysis (LSA) (Rao and Kak 2011), and Vector Space Model (VSM) (Zhou et al. 2012). The IR approach to bug localization generally consists of treating source files as documents, against which a query, represented by the bug report, will be run. Source files that share more terms with the bug report are ranked as having a higher probability of containing the bug. Recently, *structured* information retrieval has also been used for bug localization (Saha et al. 2013; Wang and Lo 2014). Structured IR-based techniques take advantage of the known structure of source files to calculate textual similarity. This allows the technique to place different emphasis on different program constructs, such as class and variable names. Bug localization techniques based on structured information retrieval, such as BLUiR (Saha et al. 2013) and AmaLgam (Wang and Lo 2014), have shown considerable improvements over other traditional IR approaches (Section 2.2). These (Saha et al. 2013; Wang and Lo 2014) are among the best-performing IR-based bug localization techniques available, although their effectiveness have been evaluated in only four projects[1].

In structured IR, source files are not simple text documents. As the structure of source files is considered, bug localization techniques based on structured IR have to be designed for specific languages. Java is the language that is commonly used to assess the effectiveness of new techniques, including BLUiR and AmaLgam. The repeated use of the same language for evaluation of bug localization techniques has the advantage of allowing a direct comparison of the results. Nonetheless, there is a need to evaluate successful bug localization techniques using different languages. Since structured IR leverages syntactic features of the source code, the usage of different types of program constructs may be able to improve bug localization effectiveness.

To investigate how different combinations of program constructs affect bug localization effectiveness, we have evaluated BLUiR, BLUiR+ (a variation of BLUiR), and AmaLgam on 20 C# projects available on GitHub. C# is a general-purpose, object-oriented language that shares many traits with the Java language, but also have some distinct features. For example, some C# constructs, like properties and structures ("structs") are inexistent in Java. C# is a popular language (TIOBE Software BV 2016) and is present among the top 10 languages in number of GitHub repositories (The GitHub Blog 2015). This growing popularity justifies the need for experiments using the C# language, in addition to the available studies reported from Java projects. Evaluating these techniques on C# projects has provided an assessment on how the techniques behave with a different object-oriented programming language.

Additionally, we have addressed some shortcomings from previous studies on IR-based bug localization. In many studies, as Rao and Kak (2013a) point out, researchers have merely chosen a single version of the project and run the localization algorithm for all available bug reports on that same version. However, a rigorous evaluation of bug localization effectiveness should consider the appropriate project version for every bug report under analysis (Rao and Kak 2013a), i.e., the version where the bug actually occurred. Without this step, there is a high chance that the bug is not even present on the code being analyzed.

Another shortcoming addressed in our study involves what Kochhar et al. (2014) call *localized bug reports*, i.e., bug reports that already mention the file containing the defect on its own description. These bug reports should not be considered in the evaluation of bug localization techniques because (i) they artificially increase the effectiveness of the techniques (Kochhar et al. 2014), and (ii) it is unlikely that developers would even need assistance from an automated technique to localize these bugs.

The contribution of each program construct type is a question still not answered by previous studies. The key success factor for an IR-based bug localization technique lies on its ability to match terms from bug reports and source files effectively. Consequently, it becomes important to understand in more depth how these constructs individually contribute to bug localization. This knowledge would enable us to attribute higher weights to the most contributing constructs, as well as discard low contributing ones, possibly increasing effectiveness. Previous studies do not assess the contribution of particular construct types to the overall effectiveness of bug localization techniques. They often assume only key constructs in object-oriented programming languages, such as classes and methods, should be explicitly considered in their underlying localization models (Saha et al. 2013; Wang and Lo 2014; Zhou et al. 2012).

In this article, we extend our previous study (Garnier and Garcia 2016) by analyzing the contribution of specific program constructs to bug localization effectiveness. For such, we apply Principal Component Analysis (PCA), a statistical procedure which may be used to reduce the dimensionality of a dataset, on results obtained from our AmaLgam extension (Garnier and Garcia 2016). Results from this analysis point to *Methods* and *Classes* as the constructs that contribute most to the effectiveness of bug localization, whereas *Interfaces* were less correlated to effectively located bug reports. The analysis also revealed that *Properties* and *String literals*, constructs not considered in previous studies, contribute significantly to bug localization effectiveness. Hence, these constructs should not be left out from future bug localization models. Identifying the most relevant constructs have allowed us to experiment even further by enabling the specification of different weights for each construct. Our results show that changing the emphasis on particular constructs by fine-tuning these weights may increase the effectiveness of bug localization.

Next section introduces concepts and related work on bug localization. Section 3 presents our study (Garnier and Garcia 2016) on the effectiveness of BLUiR (Saha et al. 2013), BLUiR+ (Saha et al. 2013), and AmaLgam (Wang and Lo 2014) on C# projects. In Section 4, we evaluate the contribution of C# constructs to bug localization effectiveness. Section 5 concludes our paper.

2 Background

2.1 Information retrieval

Many state-of-the-art bug localization techniques are based on information retrieval. Information retrieval (IR) consists of finding documents within a collection that match a search query (Manning et al. 2008). When applying IR to bug localization, source code files become the collection of documents, and the bug report represents the query. Then, the task of finding buggy files is reduced to the IR problem of determining the relevance of a document to a query. The relevance is determined by preprocessing the query and the set of documents and then calculating the similarity between each document and the query.

The preprocessing consists of three steps: text normalization, stopword removal, and stemming. Text normalization extracts a list of terms that represents the documents and the query, by removing punctuation marks, performing case folding, and splitting identifiers. After normalization, common English stopwords are removed from the list of terms. Finally, stemming converts each term to a common root form, to improve term matching by representing similar words with the same term.

After preprocessing, the similarity between documents and the query must be calculated. A common approach is based on vector space model (VSM), where each document is expressed as a vector of term weights. These weights are typically the product of term frequency and inverse document frequency (TF-IDF) of each term. Given two documents, their similarity can be measured by computing the cosine similarity (Baeza-Yates and Ribeiro-Neto 1999) of their vector representations.

Information retrieval results are usually presented as a list of documents sorted by relevance (similarity) to the query. It is often enough to present a small set of documents that a user can browse to locate the needed information. For this reason, IR models are commonly evaluated by their ability to retrieve relevant documents at the first N positions of a list using Top-N or Hit@N metrics. Another widely used metric for the effectiveness of IR models is the mean average precision (MAP). MAP provides a single-figure measure of the quality of information retrieval when a query may have multiple relevant documents (Manning et al. 2008).

2.2 Bug localization

A tendency observed in some recent works on bug localization is the combination of distinct sources of information to improve effectiveness. BugLocator (Zhou et al. 2012) is such an example. Given a bug report, it uses information retrieval to find relevant source files by comparing the bug report with source files and with previous bug reports. In the latter case, the authors assume that to fix similar bugs, developers tend to modify similar sets of files. BugLocator can combine both sources of information (i.e., similarity data from source files and previous bug reports) according to a given weight. The authors evaluated various combinations, and found that BugLocator works best when bug similarity data is weighted between 20% and 30%.

Saha et al. (2013) developed BLUiR (Bug Localization Using information Retrieval), a bug localization technique based on the concept of structured information retrieval. In structured IR, fields from a bug report and code constructs, such as class or method names, are separately modeled as distinct documents. Consequently, bug reports and source files are not counted as single documents. Instead, BLUiR breaks bug reports into summary and description, while source files are split into class names, method names, variable names, and comments. Each part of a bug report is compared to each part from the source file. The similarity of a bug report and a source file is given by the sum of the calculated similarities.

Saha et al. also created a variant of BLUiR, called BLUiR+ (Saha et al. 2013), which combines the approaches from BugLocator and BLUiR. In addition to the features of BLUiR, BLUiR+ also leverages information from previous similar bug reports, if available, similarly to BugLocator. The authors compared both variations, BLUiR and BLUiR+, to BugLocator without and with bug similarity data, respectively. The comparison showed a performance improvement regarding mean average precision (MAP) of up to 41% (24% on average) when neither BugLocator nor BLUiR used bug similarity data. When bug similarity data was used, BLUiR+ outperformed BugLocator by up to 29% (11% on average). Another interesting result is that BLUiR achieved results similar or superior to those of BugLocator even when the latter did use bug similarity data, and the former did not. This finding indicates that the structured IR approach used by BLUiR could compensate for the lack of previous bug report information.

Another example of bug localization technique that combines structured IR with other sources of information is AmaLgam (Wang and Lo 2014). AmaLgam is a technique for locating buggy files that combines the analysis of: (i) version history, (ii) bug report similarity, and (iii) structure of documents, i.e., bug reports and source code files. AmaLgam relies on the bug localization formula used in Google's bug predictor (Lewis and Ou 2011), which considers version history. AmaLgam then combines this formula with BugLocator and BLUiR. Each of the three approaches has a corresponding component in AmaLgam that calculates a single score for each file. The final score of a source file is a weighted average of the three individual scores. Experiments showed that AmaLgam achieved optimal results with weights of 30% for version history, 14% for bug report similarity, and 56% for structured IR from source files.

2.3 Structured information retrieval

The studies mentioned in Section 2.2 show that structured information retrieval, by leveraging the known structure of the documents involved in the retrieval process, has been more effective than "plain" information retrieval. The potential of structured IR motivates researchers to investigate this approach to bug localization in the context of other programming languages, such as C#. Therefore, we selected some of the best performing techniques based on structured IR by the time of our research, BLUiR, BLUiR+, and AmaLgam, for our study. We present additional details about the mechanics of these techniques as follows.

2.3.1 The BLUiR approach

The key insight of BLUiR is the use of source file structure to improve effectiveness. In a traditional IR-based approach, the entire content of both source files and bug reports is used to calculate textual similarity. BLUiR, instead, breaks source files into four parts: class names, method names, variable names, and comments. Bug reports are split into two parts: summary and description. BLUiR then calculates the similarity between each file part and bug part separately, summing the eight individual similarities in the end. The formula below represents the core of the BLUiR approach.

$$sim(file, bug) = \sum_{fp \in file} \sum_{bp \in bug} sim(fp, bp) \tag{1}$$

In the previous equation, *file* and *bug* are a source file and a bug report, and *fp* and *bp* are its respective parts. The similarity between a bug report and a source file is given by the sum of the similarities of their parts.

2.3.2 The AmaLgam approach

AmaLgam takes as input a new bug report, representing the bug to be localized in the source code, a set of source files, commit history data, and a set of older bug reports. AmaLgam has three components that produce suspiciousness scores for each source file, based on different sources of information. Individual scores are then combined by a fourth component (composer) into a single score for each source file. AmaLgam components are described below.

Version history component. This component consists of Google's adaptation (Lewis and Ou 2011) to the algorithm from Rahman et al. (2011). The adaptation also includes a further modification: a parameter (k) that restricts the version history period (in days) to be considered. It was observed by Wang and Lo that considering only more recent commits provided a good trade-off between precision and performance. The optimal value found by the authors was $k = 15$ (Wang and Lo 2014).

Report similarity component. AmaLgam's report similarity component is based on the SimiScore formula from BugLocator (Zhou et al. 2012), also used by BLUiR+ (Saha et al. 2013). The component considers the textual similarity between bug reports and the number of files modified to fix each bug report. The assumption is that to fix similar bugs, developers tend to modify similar sets of files.

Structure and Composer components. AmaLgam's structure component uses the approach of the BLUiR technique (Saha et al. 2013). Then, the composer component takes the scores produced by the three other components and combines them into a final suspiciousness score. It first combines the results from the report similarity (*scoreR*) and structure (*scoreS*) components. This result is then combined with the score from version history component (*scoreH*), according to the following equations:

$$scoreSR(f) = (1 - a) \times scoreS(f) + a \times scoreR(f) \tag{2}$$

$$scoreSRH(f) = \begin{cases} (1 - b) \times scoreSR(f) + b \times scoreH(f), & scoreSR > 0 \\ 0, & otherwise \end{cases} \tag{3}$$

Parameters a and b in the equations above determine the weight of the contribution of each component to the final suspiciousness score. Based in their own experiments and in results from Zhou et al. (2012) and Saha et al. (2013), the authors adopted the default values of 0.2 for parameter a and 0.3 for parameter b (Wang and Lo 2014).

3 Evaluation of bug localization techniques

In this section, we investigate how the usage of different sets of program constructs influences the effectiveness of bug localization. For such, we evaluate BLUiR (Saha et al. 2013), BLUiR+ (Saha et al. 2013), and AmaLgam (Wang and Lo 2014) on 20 C# projects. C# is a popular language (TIOBE Software BV 2016; The GitHub Blog 2015), similar to Java, although with significant differences, especially regarding the available constructs. The similarity will allow us to draw a parallel with Java results. At the same time, the differences will allow us to explore constructs inexistent in Java, such as properties and structures.

We also discuss dataset preparation steps conducted to mitigate shortcomings from previous studies (Section 3.4). These preparation steps include selection of appropriate project versions, removal of bug reports that could influence the evaluation, and removal of test files from the search scope. Results show that, with the appropriate data preparation steps, effectiveness of bug localization is at least 34% lower, compared to the effectiveness without the data preparation steps (Section 3.5).

After evaluating the techniques as they were conceived, we adapt them in order to assess their sensitivity to the consideration of more constructs (Section 3.6). We define three construct mapping modes, which represent different forms of splitting source files, namely, *Default*, *Complete*, and *Mixed* modes. The *Default* mode corresponds to the same mapping used by BLUiR (Saha et al. 2013), BLUiR+ (Saha et al. 2013), and AmaLgam (Wang and Lo 2014), where source files are splitted into four documents, consisting of class names, method names, variable names, and comments. In the *Complete* mode, all the available constructs are considered separately and every source file is splitted in 12 parts, each one corresponding to each available C# construct. Finally, the *Mixed* mode maps all C# constructs into four groups, similarly to the original mapping used by BLUiR and AmaLgam. The *Mixed* and the *Complete* construct mapping modes were able to increase bug localization effectiveness by 8% and 18%, on average (Section 3.7).

3.1 Research questions

In order to perform a realistic evaluation of bug localization techniques based on structured IR, we need to evaluate them on different scenarios. An initial step in that direction is to understand the behavior of prominent bug localization techniques applied to an object-oriented programming language that is slightly different from Java. Java has been the focus of previous studies of structured IR-based techniques for bug localization (Section 2.3). Nonetheless, software engineers remain unaware to what extent they can rely on these techniques to perform bug localization activities in projects structured with other programming languages. We have selected C# as it is a general-purpose, object-oriented language that shares many traits with the Java language, but also have some distinct programming features. For example, some widely used C# constructs, like properties and structures ("structs"), are inexistent in Java. To the best of our knowledge, neither of these techniques have been previously evaluated on other object-oriented programming languages, such as C#.

We unfold our general goal in the following research questions:

RQ1: Are BLUiR, BLUiR+, and AmaLgam effective to locate bugs in C# projects?

The effectiveness of current structured IR techniques, i.e., BLUiR, BLUiR+, and AmaLgam, have been assessed and confirmed only for Java projects. However, developers using many other languages could also benefit from such techniques. To address this gap, we ran the selected techniques in their best performing configurations (Section 2.3) on a set of C# projects. The results enabled us to address RQ1 by assessing the effectiveness of these techniques on a previously untested programming language.

RQ2: Does the addition of more program constructs increase the effectiveness of bug localization on C# projects?

To fully understand the potential of structured IR techniques, we need to analyze their sensibility to particular constructs of a programming language. Therefore, we addressed RQ2 by focusing this analysis on program constructs that were also not considered in previous studies (Saha et al. 2013; Wang and Lo 2014), such as string literals, interfaces, and enumerations. In addition, there are language features from C# that do not exist in Java, such as structures and properties. The effects of their explicit consideration on state-of-the-art bug localization are not well understood. Thus, we investigated to what extent the effectiveness of a structured IR technique would benefit from the explicit consideration of these source code constructs.

3.2 Evaluation metrics

This section describes the metrics used to assess the effectiveness of the techniques on selected projects. We focused on the use of two sets of metrics typically used in recent studies (Saha et al. 2013; Wang and Lo 2014; Zhou et al. 2012):

Top-N and Hit@N: Indicates the percentage of bug reports that have at least one buggy file ranked by the technique in the top N positions. Typical values for N are 1, 5, and 10 (Saha et al. 2013; Wang and Lo 2014; Zhou et al. 2012).

Mean average precision (MAP): This metric considers the ranks of all the buggy files, not only the first one. It is computed by taking the mean of the average precision scores across all queries.

To measure the effectiveness of the technique for a given project, the arithmetic mean of the results for each bug is taken. Finally, the effectiveness of a technique, or one of its variations, corresponds to the average of the results for each project. We use commit and bug report data obtained from the selected projects (Section 3.3) as the oracle against which we compare the results of our implementation. When a bug report explicitly contains a link to a commit, we consider the files modified in the commit as the ones that solved the bug. This is a common assumption in many bug localization studies (Saha et al. 2013; Wang and Lo 2014; Zhou et al. 2012). When there is no explicit link between a bug report and a commit, we use conventional heuristics (Bachmann and Bernstein 2009) to infer this relationship. These heuristics consist of looking for commits that contain messages such as *"Fixes issue 97"* or *"Closes #125"*, which usually denotes the ID number of the associated bug report.

3.3 Project selection

For our experiment, we needed a number of C# projects with available information on their source code, commits, and bug reports. We could not find a bug dataset for C# projects, like iBUGS (Dallmeier and Zimmermann 2016) or moreBugs (Rao and Kak 2013b). Then, we used GitHub search functionality[2] to obtain a list of large C# projects, by searching for projects with 1000 or more stars and 100 or more forks. These parameters indirectly allowed us to satisfy the requirement for large projects. The query returned almost 80 projects from various domains, including development tools, compilers, frameworks, and games.

We used GitHub API to download commit and issue data from the projects. We downloaded the 1000 most recent issues for each project[3], and then all the commits that happened within the period covered by the issues. Next, we processed the data to identify (i) issues that could be characterized as bugs and (ii) files modified to fix the bug. For characterizing a GitHub issue as a bug, we relied on the labels applied by the users. Issues with at least one label containing terms such as "bug" or "defect" were considered a bug report. As for the files modified to fix a bug, they are determined by the associated commit, as explained in Section 3.2. Since we are focusing on C# code, we excluded from evaluation bugs that did not touch at least one C# file.

After processing downloaded data, only those projects where we could find at least 10 bugs whose resolution modified at least one C# file were kept for the experiment. We processed the projects in the order returned by the query until we reached 20 projects that met our selection criteria. Table 1 presents a comparison between the dataset of C# projects used in our study and the dataset of Java projects used in recent studies (Saha et al. 2013; Wang and Lo 2014; Zhou et al. 2012) of the same techniques.

In our dataset, about 61% of the files contained in the repositories were C# files, the ones we actually used to search for bugs. This happens because many files represent: (i) configuration or HTML files, or (ii) source files structured with other programming languages in case of multi-language projects. Actually, the existence of multi-language projects also highlights the importance of evaluating bug localization techniques in different programming languages. For the Java dataset, it was not clear whether the total referred only to Java source files or to all repository files. Therefore, we assumed the latter.

As for the bugs, all of them are treated as issues in GitHub issue tracker, although not all issues are bugs. After downloading all the available issues, we associated them with a commit whenever possible, using the criteria explained in Section 3.2. This step reduced the number of available bugs to 17% of the original issue count. Then we discarded issues that were not labeled as "bug", which reduced the number of available bug reports even further, down to 878 (5% of the initial number of issues).

3.4 Dataset preparation

As mentioned in the Introduction, previous studies suffer from a series of shortcomings regarding their experimental setup. Next, we describe how we handled these shortcomings in our evaluation.

3.4.1 Version selection

Previous studies on bug localization commonly selected only a single release and ran the bug localization for all bugs on the same release. Results reported in this manner cannot

Table 1 Dataset comparison

Dataset details	Java	C#
Projects	4	20
Files	N/A	46,752
Source files	20,223	28,596
Issues	N/A	16,630
Traceable to commits	N/A	2839
Classified as bug	3479	878

be fully trusted (Rao and Kak 2013a), because there is a high chance that the bug is not even present in the code being analyzed. To overcome this problem, we identified the version of the source code that was active by the time the bug was reported by searching for the oldest commit that happened before the bug report creation. The source code for every identified version was downloaded, and each bug was localized on its corresponding version.

3.4.2 Bug report selection

Some bug reports already inform the location of the defect in the source code, by mentioning the file where the bug was observed. Kochhar et al. (2014) demonstrated that including these bug reports on the evaluation of a bug localization technique significantly influences the results by artificially increasing the reported effectiveness. The authors classified bug reports in three categories: *fully localized*, *partially localized*, and *not localized*, which mean that a bug report mentions *all*, *some* or *none* of the files modified to fix a bug; respectively. We removed fully and partially localized bug reports from our evaluation, meaning that we included only those bug reports that contained no mention of the faulty files. Although this step contributes to more realistic results, it reduced the number of available bug reports in 51%, from the 878 reported in Table 1 to 450 (3% of the initial issue count).

3.4.3 Source file selection

Software projects often include test code. Test code may contain bugs, which, in theory, may be reported just like production code. However, bug localization algorithms should not include test code within their scope. Consider, for instance, three bug reports, whose resolution involved the modification of (i) only production code (no test code); (ii) production and test code; and (iii) only test code. In the first case, it is obvious that localization does not benefit from considering test code. When the resolution of a bug requires changing production and test code (second case), it is usually because a test was added or modified to catch the referred bug in the future. Test code was not the *source* of the failure, though. Therefore, modified test files are not what developers expect as an answer from the localization algorithm in this case. Finally, when a bug in the test code itself is caught (third case), developers already have detailed information provided by the test framework, which includes the location of the bug. Thus, even if a developer chooses to report a test bug instead of fixing it immediately, it is likely that this report will include the detailed information already provided by the test framework. Therefore, bug reports on test code are rarer (because the developer may choose to fix the bug instead of reporting it) and likely to be localized (because test frameworks already indicate the buggy files). This rationale led us to restrict the localization to production code.

We excluded test files from the scope of the analysis by ignoring all files that contain the word “test” on its path. We confirmed with manual inspection on two sample projects that this simple heuristic was able to accurately remove the undesired files, since it reflects (in our sample) the common developer practices of naming test files with a “Test” prefix of suffix, or placing test code in a separate directory named “test”.

3.5 Effectiveness of structured IR-based bug localization in C# projects

3.5.1 Effectiveness without dataset preparation.

Initially, we questioned whether current state-of-the-art bug localization techniques based on structured information retrieval, i.e., BLUiR, BLUiR+, and AmaLgam, would effectively locate bugs in C# projects. Considering results for Java, one would expect similar levels of effectiveness for C# as well, given the apparent similarities between the languages. To answer our first research question, we ran the bug localization techniques on downloaded projects using the reported optimal configuration for each technique (Section 2.3).

For the sake of comparison, we initially run the algorithms *without* the preparation steps discussed in Section 3.4. Then, we repeated the evaluation of the three techniques *including* these steps. For each technique, we took the average MAP, which consists of the arithmetic mean of the MAPs from each project. Table 2 presents the average MAP values observed for the set of evaluated projects alongside the same measure from Java projects. Considering all the techniques, the average MAP achieved by each technique with C# projects was around 0.307.

Opposed to the previous findings in Java projects, the selected bug localization techniques showed lower effectiveness in terms of average MAP. This result should be interpreted carefully, as the projects are different and cannot be compared. Nevertheless, the observed variation is explainable in part due to the higher number of projects analyzed: 4 in the Java studies (Saha et al. 2013; Wang and Lo 2014) against 20 in our C# study. Within projects in the same language, the techniques presented similar behavior: AmaLgam performed better than BLUiR+, which outperformed BLUiR. Recall from Section 2.2 that each technique uses a superset of the information used by the previously proposed technique: BLUiR is based on the similarity of bug reports and source code, BLUiR+ adds the similarity of previous bug reports to the equation, while AmaLgam also considers version history. This could be considered an indication that, in fact, hybrid techniques which combine additional sources of information tend to perform better than their predecessors do.

Table 2 also presents highest and lowest MAP scores for each technique and language. The minimum MAPs from the C# group were lower than minimum values from the Java group for all three techniques. The maximum MAPs, on the other hand, were similar. In fact, there was one project where the techniques reached even higher MAP values, but it was considered an outlier. Average MAPs for the outlier were 0.770, 0.767, and 0.747 for BLUiR, BLUiR+, and AmaLgam, respectively (Fig. 1).

In spite of the lower averages relative to the Java evaluation, six C# projects still have attained MAP scores superior to the average of their Java counterparts in at least one technique. This implies that, in principle, there is no impediment to the usage of bug localization on C# projects due to features of the language itself, leaving room for the investigation of alternatives to increase the effectiveness of the techniques. In Section 3.7 we propose such alternatives by evaluating the effects of different mappings of language constructs on the bug localization algorithm. However, we must first generate an accurate baseline against which these alternatives can be compared.

Table 2 C# and Java results—MAP

Technique	Minimum MAP		Average MAP		Maximum MAP	
	Java	C#	Java	C#	Java	C#
BLUiR	0.24	0.103	0.38	0.302	0.56	0.596
BLUiR+	0.25	0.125	0.39	0.307	0.58	0.596
AmaLgam	0.33	0.120	0.43	0.312	0.62	0.604

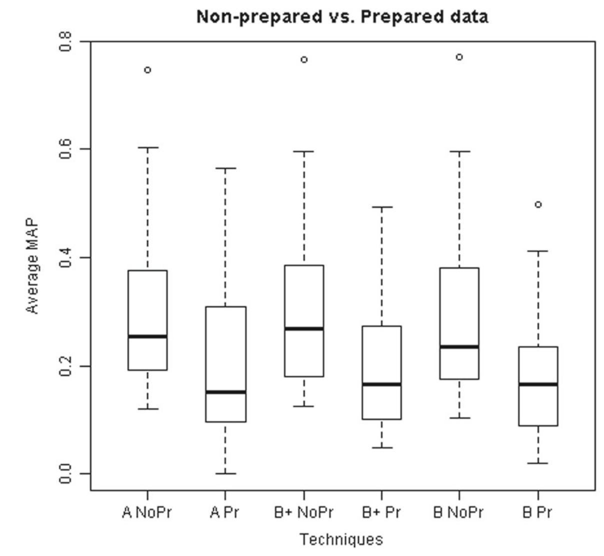

Fig. 1 Effectiveness of techniques with C# projects—Non-prepared vs. prepared data

3.5.2 Effectiveness with dataset preparation.

To generate more realistic results, we have also evaluated the three techniques including the dataset preparation steps presented in Section 3.4. Table 3 presents minimum, maximum, and average values for each technique when the dataset was properly prepared, and the decrease relative to results with no preparation steps.

Wilcoxon Signed-Rank tests[4] with 95% confidence level confirmed that additional preparation steps on the dataset significantly decreased the MAP scores for the three techniques. Complete details about the tests, including p-values, are available at the study website (Garnier 2016). The maximum values indicate that some projects were still able to achieve reasonable scores. However, compared to the execution with no preparation steps, the effectiveness of all projects decreased, on average, more than 30% for all the evaluated techniques. Figure 1 presents a graphical comparison of the effectiveness with and without the preparation steps. It becomes clear from the data that bug localization studies must not ignore these steps, under the penalty of reporting results incorrectly higher than what would be found in actual settings.

Removing localized bug reports from this kind of evaluation has indeed practical importance, as non-localized bug reports are exactly the kind of report where developers would need the assistance of a localization technique. Therefore, the effectiveness of IR-based bug localization in terms of mean average precision can still be considered too low for these techniques to be applied in practice.

Table 3 Effect of dataset preparation steps on bug localization – MAP

Technique	Minimum	Average	Maximum
BLUiR	0.020	0.183 (-40%)	0.499
BLUiR+	0.048	0.198 (-36%)	0.493
AmaLgam	0.044	0.206 (-34%)	0.565

On the other hand, the expectations for this kind of technique must also be put into context. No matter how effective they are, bug localization techniques do not eliminate the need for the developer to examine and fix the buggy file. Therefore, instead of pinpointing the exact files where the bug is located, it may be acceptable for the technique to provide a list featuring a few candidates. Table 4 shows that the best performing technique—AmaLgam—was able to return a buggy file at the top of the list 20% of the times, and in 57% of the times there was a buggy file among the 10 first files returned by the technique. Analyzing hundreds of files and correctly placing at least one buggy file in a list of 10 candidates for almost 60% of the time is a reasonable result.

Nevertheless, these results reinforce that there is still room for improvement. As discussed in Section 2.3, structured information retrieval is the component that contributes the most to the effectiveness of state-of-the-art bug localization techniques (Saha et al. 2013; Wang and Lo 2014). Thus, we extended the underlying algorithm of AmaLgam's structure component (Section 2.3) to assess its effectiveness when using a different set of programming language constructs. We present the results in the next section.

3.6 Model adaptation

Structured IR demands the extraction of identifiers from source code. For this task, we used the .NET Compiler Platform (Microsoft 2014). As C# is an object-oriented language, similar to Java, it has the same four constructs considered on BLUiR's original evaluation: classes, methods, variables, and comments. However, C# also has constructs that either were not considered by BLUiR (and, consequently, neither by BLUiR+ nor AmaLgam) or do not exist in Java. Table 5 summarizes these differences.

We do not consider language keywords (e.g., "if", "for", "while") as they would carry little meaning for bug localization purposes. Therefore, we limit the considered constructs to identifiers (Table 5). Because identifier constructs can be named, they are more likely to reflect domain-specific concepts, which increases the possibility of matching bug report terms, justifying their choice.

BLUiR breaks a source file into parts. Each part contains identifiers from one kind of construct. To deal with the different kinds of constructs, while keeping the underlying philosophy of BLUiR, we devised three alternative configuration modes to run the experiment:

- *Default:* Strictly uses only the same constructs used by BLUiR, ignoring any other construct.
- *Complete:* Uses all the constructs, with each one mapped to an exclusive file part.
- *Mixed:* All constructs are used, but they are mapped to one of the four file parts corresponding to the constructs originally used by BLUiR.

Default mode is used as a baseline for the sake of comparing our results with the original evaluation in Java projects (Saha et al. 2013). *Complete* mode represents the simplest way of including new constructs in BLUiR's algorithm. *Mixed* mode represents an alternate

Table 4 AmaLgam effectiveness with dataset preparation steps

Technique	Hit@1	Hit@5	Hit@10
AmaLgam	20%	46%	57%

Table 5 List of C# constructs

C# construct	Equivalent in Java?	Considered by BLUiR?
Classes	Yes	Yes
Comments	Yes	Yes
Enumerations	Yes	No
Fields	Yes	No
Interfaces	Yes	No
Methods	Yes	Yes
Namespaces	Yes (packages)	No
Parameters	Yes	No
Properties	No	No
String literals	Yes	No
Structures	No	No
Variables	Yes	Yes

way of computing new constructs, by mapping them to one of the preexisting categories. For example, interfaces, structures, and enumerations are semantically close to classes. Therefore, for the purpose of bug localization, it could be enough to consider code elements of any of these types as "classes". In a similar vein, string literals usually represent plain text inserted into source files, such as comments do. Therefore, string literals and comments could be mapped together in the same file part.

The difference between some constructs is negligible in practice. For instance, variables and parameters are distinct constructs, strictly speaking. However, from the developers' point of view, they are both handled as variables. Although it was not clear, this simplification might have been used in the BLUiR evaluation. Thus, the *Mixed* mode addresses this possible ambiguity, by defining a broader interpretation to the four constructs mentioned by Saha et al. (2013). The mapping strategy for each mode is shown in Table 6.

3.7 Usage of more constructs to improve bug localization effectiveness

The set of constructs used by BLUiR, BLUiR+, and AmaLgam includes basic constructs from object-oriented languages (classes and methods) and constructs from programming languages in general (variables and comments). However, some subtleties about construct selection were omitted or unaddressed in previous studies. There are additional types of constructs that could be explicitly considered by bug localization techniques. When considering a different programming language, with a different set of constructs, this issue becomes more relevant.

To answer whether the consideration of more construct types could improve the effectiveness of bug localization, we designed the three construct-mapping modes described in Table 6. Next, we needed to select a technique to adapt according to the new mapping modes. As the focus of the adaptations are in the usage of the program constructs by the

Table 6 Construct-mapping strategies

Mode	Construct mapping
Default	Classes, methods, variables and comments (one file part for each)
Complete	All constructs (one file part for each)
Mixed	Part 1: Classes, enumerations, interfaces, namespaces, and structures Part 2: Methods Part 3: Fields, parameters, properties, and variables Part 4: Comments and string literals

structure component, all three techniques—BLUiR, BLUiR+, and AmaLgam—were candidates. Moreover, the difference of BLUiR and BLUiR+ to AmaLgam is that the latter also analyzes additional components (Version History and Report Similarity components), which do not use program constructs at all. Therefore, regarding the contribution of program constructs (focus of our study), one can assume the same results regardless of our choice between BLUiR, BLUiR+, or AmaLgam. Thus, we selected AmaLgam, the best performing technique according to the evaluation from Section 3.5.2, adapted it to use the three mentioned modes, and applied it to the set of C# projects. We present the average MAPs (Table 7) and a box plot (Fig. 2) summarizing the performance observed for each mode.

The usage of all the 12 constructs associated with the *Complete* mode increased the average MAP of AmaLgam to 0.244, an increase of 18%. *Mixed* mode, which also uses the 12 constructs but maps them into four categories (Table 6), showed a smaller increase on average, to 0.222 (near 8%). From these results, only the improvement associated with *Complete* mode was statistically significant, according to Wilcoxon Signed-Rank tests with 95% confidence level (Garnier 2016). The effect of the three construct-mapping modes on individual projects was generally the same observed on average values: the higher increase was associated with the *Complete* mode, while *Mixed* mode caused a more modest increase, as shown in Fig. 2.

The reason why *Complete* mode was able to produce better results can be explained by BLUiR formula (also used by AmaLgam) for determining the similarity of a bug report and a source file (Section 2.3), which involves the summation of the similarities of all pairs of bug-file parts. In *Default* mode, the total number of similarities to be summed is 8—two parts from the bug report multiplied by four parts from the source code files. In *Complete* mode, the number of "similarities" increases to 24 (12 file parts $\times$ 2 bug parts). Similarity results are normalized before the rank of files is generated, such that file scores are always between 0 and 1. Therefore, the higher value that would result from the summation of more terms in *Complete* mode is unlikely to be the reason this mode produced better results. In addition, the *Mixed* mode restricts the number of terms to be added up to 8, similarly to the *Default* mode. Since the *Mixed* mode has also produced results higher than those of the *Default* mode, we conclude that the consideration of more source constructs by itself contributed to increasing the bug localization effectiveness.

Execution time. As we logged all the bug localization runs, including the time elapsed for each run, we can assert that the modes that process more constructs (i.e., *Mixed* and *Complete* modes) *do* take longer to complete than the *Default* mode. However, in the first runs of the adapted modes, we perceived no significant increase in the elapsed time. Thus, we did not design a research question to evaluate the variation of the elapsed time for each bug localization run. Therefore, we did not implement measures to control the execution environment, such as avoiding parallel execution of other programs that could compromise any measurement of the elapsed time. Consequently, the data we collected

Table 7 Effectiveness of AmaLgam using different construct-mapping modes

Mode	Default	Complete	Mixed
Average	0.206	0.244	0.222

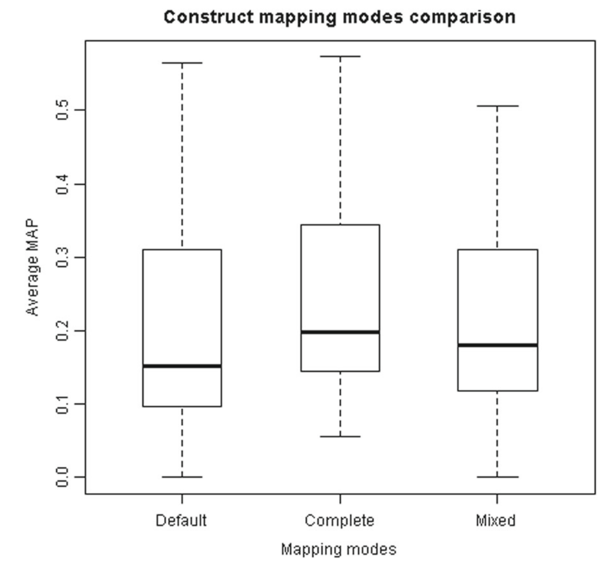

Fig. 2 Effectiveness of construct mapping modes

regarding elapsed time cannot be used to compare the cost-benefit of each mapping mode formally.

3.8 Threats to validity

3.8.1 Construct validity

As we could not find an available bug benchmark for C# projects, we downloaded issues from GitHub and used the existence of a user-applied "*bug*" label as a criterion to identify bugs among those issues. Even following this procedure, we are still subject to misclassified issues, since not all bug reports could be manually verified. We also assume all files touched by a bug-fixing commit are related to the bug. Although this is a common assumption in bug localization studies (Saha et al. 2013; Wang and Lo 2014; Zhou et al. 2012), it is possible that some files included in a commit are not actually related to the bug it fixes.

However, Kochhar et al. suggest that even though these biases (i.e., misclassified issues and files unrelated to the bug in a commit) may influence bug localization results, the influence from these particular biases is neither *significant* nor *substantial* (Kochhar et al. 2014). Thus, we concentrated on the issue of *localized bug reports*—the type of bias that, according to Kochhar et al. (2014), *do* influence bug localization results. We handled this issue by excluding localized bug reports from the analysis. By performing this exclusion, we remove an important bias that may have led previous studies to unrealistic results.

Studies involving retrospective evaluation of bugs should consider the version of the software at the time the bug was found. We addressed this issue by performing the localization on latest version available before the creation of each bug report. Strictly speaking, this does not guarantee that the selected version actually contains the bug reported. However, the selection of a previous version of the code for each bug report is a close approximation. Moreover, this step mitigates a threat ignored in many recent studies on bug localization, including those from Saha et al. (2013) and from Wang and Lo (2014).

Finally, the presence of test files also influences bug localization results, since bug reports related to these files are very likely to be localized (Section 3.4.3). We eliminated this bias by excluding test files from the evaluation. The full rationale for performing these exclusions was discussed in Sections 3.4 and 3.5.2. The adoption of these dataset preparation steps argues for a strong construct validity in our study.

3.8.2 External validity

In an attempt to increase generalizability, we selected a higher number of projects compared to previous studies. Given the criteria defined in Section 3.3, we were able to select 20 projects, a considerably higher number of projects compared to other studies on bug localization (5 times more than the BLUiR (Saha et al. 2013), BugLocator (Zhou et al. 2012), and AmaLgam (Wang and Lo 2014) studies). The absence of a standardized bug database, however, greatly reduced the amount of bug reports available for the experiment (Table 1). The relatively low quantity of bug reports and the variation in quantity and quality of bug reports observed on each project are threats to the external validity of our study. However, we consider that bug localization techniques must be assessed under realistic project settings, i.e., the quantity of bug projects naturally varies widely from one project to another. The complete list of evaluated projects and the number of bug reports evaluated for each one is available at the study website (Garnier 2016).

Another threat is the fact that all selected projects were open-source. This kind of project has a characteristic workflow that differs from that found on proprietary projects. Different policies on bug reporting, for example, may significantly influence the results of bug localization techniques. Therefore, the results presented in this study are only representative of the workflow typically practiced in open-source projects.

4 Analysis of the contribution of program constructs to bug localization

Structured information retrieval (IR) has been able to increase the effectiveness of static bug localization techniques (Saha et al. 2013; Wang and Lo 2014). The key feature of structured IR-based techniques refers to how they break up source files, based on constructs available in the adopted programming language. Bug localization techniques based on traditional IR calculate the similarity between a source file and a bug report considering the whole file (Section 2.2). Conversely, structured IR-based techniques break source files into multiple parts, one for each construct recognized by the technique (Section 2.3). Each of these parts contains only terms that are instances of the corresponding construct in the original source file. Then, instead of calculating similarity using the whole file, the final similarity between a bug report and a source file is the sum of the similarities between each part of the source file and the bug report (Equation 1). BLUiR (Saha et al. 2013) recognizes four Java constructs: class names, method names, variable names, and comments (Section 2.3.1). Thus, it breaks source files into four parts. The same approach is followed by BLUiR+ (Saha et al. 2013) (Section 2.3.1) and AmaLgam (Wang and Lo 2014) (Section 2.3.2).

However, structured IR has not been thoroughly explored yet. In addition to the limitation of being evaluated only on four projects, the original models of BLUiR (Saha et al. 2013), BLUiR+ (Saha et al. 2013), and AmaLgam (Wang and Lo 2014) used only four constructs from the Java language (Section 3.6). Thus, it is unknown whether other constructs, such as interfaces or enumerations, could have influenced bug localization results. This question becomes even more relevant when source files are written in other programming languages, such as C#, which supports constructs inexistent in Java (Table 5).

In this section, we investigate the influence of different program constructs on the effectiveness of structured IR-based bug localization. In this investigation, we use results obtained with the *Complete* mode (Section 3.6), as this construct mapping mode increased bug localization effectiveness by including all available C# constructs into the localization process (Section 3.7). Then, we use a statistical procedure called Principal Component Analysis (PCA) to quantify the contribution of each construct to the similarity score attributed to source files (Section 4.1). This analysis will reveal the extent of the correlation between those constructs and bug localization results.

Finally, we explore the different contributions from each construct by further modifying the bug localization algorithm. First, we evaluate whether suppressing low-contributing constructs influences the result either positively or negatively (Section 4.2.2). Next, we evaluate whether bug localization effectiveness can be increased by attributing higher weights the most influential constructs in the file score equation (Equation 1), thus emphasizing their contribution (Section 4.2.3).

4.1 Assessment of construct contribution

As the usage of more constructs has been shown to increase bug localization effectiveness (Section 3.7), we need to understand the contributions of each construct to the effectiveness increase. For this purpose, we have devised the following question:

RQ3: Which program constructs contribute most to the effectiveness of bug localization in C# projects?

To answer RQ3, we must analyze in more depth data related to files that were effectively located. Therefore, we selected those files that were both buggy according to the oracle (Section 3.2) and highly-ranked by the technique, i.e., ranked among the top 10 positions. The number of positions, 10, is consistent with the Hit@10 metric (Section 3.2). Next subsection presents a brief description of the procedure used to assess construct contribution.

4.1.1 Principal component analysis

To answer RQ3, we use Principal Component Analysis (PCA). PCA is a statistical procedure that transforms a number of possibly correlated variables into a smaller number of variables called principal components (Jolliffe 2002). The central idea of principal component analysis (PCA) is to reduce the dimensionality of a dataset consisting of a large number of variables while retaining as much as possible of the variation present in the dataset. This is achieved by transforming to a new set of variables, the principal components (PCs), which are ordered so that the first few retain most of the variation present in all of the original variables (Jolliffe 2002). Translating bug localization domain to PCA, the constructs are the variables, and the similarity scores are the variable values.

The reasoning for using PCA to assess the contributions of constructs is that such analysis may shed light on how to further improve bug localization effectiveness. For instance, its use may indicate that the studied techniques may be more sensitive to a specific construct subset. If this is true, most influential constructs will emerge as highly correlated with the first few principal components (PCs). Furthermore, it is expected that the influence exerted by these constructs could be exploited to increase bug localization effectiveness.

4.1.2 Analysis setup

To perform the analysis, we organize relevant data in the form of a table: variables are laid out in columns, while rows correspond to data points. The 12 C# constructs (Table 5) are the variables. The data points correspond to every buggy file effectively located. The values for each variable are the summands that compose *scoreS*, the similarity score attributed by AmaLgam's structure component (Section 2.3). *scoreS* is the summation of the similarities of each pair of file and bug parts (Equation 1). Thus, for each construct, there is a term from *scoreS* that reflects its specific contribution to the structural similarity score. These are the variable values used as input for the PCA. These values were taken from the best performing mode of construct mapping (i.e., the *Complete* mode, Section 3.6). After applying the selection criteria for effective instances (i.e., buggy files ranked among the top 10 positions), 363 data points were selected to compose the PCA input.

4.1.3 Variances of principal components

PCA transforms the input data into a coordinate system such that the highest variance lies on the axis corresponding to the first principal component. Remaining components represent dimensions that account for a decreasing amount of variance. In other words, the first components explain most of the variance of the data. In our context, we explore PCA to understand which constructs better explain the data variance on bug localization results.

Figure 3 presents the degree of variance explained by each of the 12 PCs. While X-axis represents the PCs, Y-axis indicates the percentage of explained variance. Figure 3 shows that the first principal component (PC1) accounts for 20% of the variance in the data, twice as much as PC2. From PC2 through PC12, the percentage of variance smoothly decreases from 10 to 4.1%.

One of the main applications of PCA is to reduce dimensionality from a dataset. This is possible when the first few components account for a high percentage of the variance. What may be considered a high percentage of variation is subjective, although the literature suggests a sensible cutoff is often in the range of 70 to 90% (Jolliffe 2002). Thus, considering the distribution presented in Fig. 3, it would be necessary to retain the seven first PCs to account for 70% of the variance. The analysis of Fig. 3 reveals that, except for PC1, all remaining components present comparable contributions to the structural similarity scores. Hence, no component can be confidently discarded due to a negligible contribution.

Although all construct types contribute to the final scores, analysis of the variances (Fig. 3) suggests that some constructs contribute more than others do. These are probably associated with PC1, which by itself accounts for 20% of the variance in the results. It remains to be investigated which constructs are associated with the first few principal components.

4.1.4 Constructs associated with principal components

The degree of relationship between original variables and principal components created by the analysis can be measured by their correlation coefficients. A positive correlation indicates that both values (original variable and PC) increase simultaneously. Therefore,

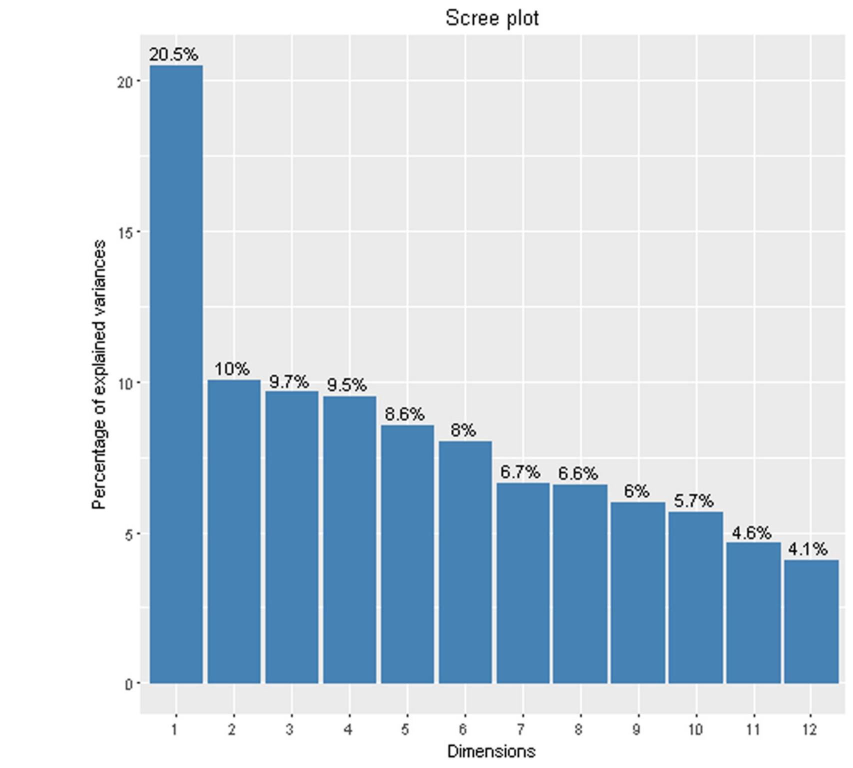

Fig. 3 Variance corresponding to each principal component

positive correlations reveal constructs that positively contribute to the result. Conversely, negative correlations indicate that while one of the values increases, the other one decreases. This situation could be interpreted as a "wrong clue" to the technique, as the negatively correlated construct would be assigning higher scores to files that, according to the rest of the constructs, should have lower scores. Therefore, constructs with a negative correlation to the PCs are likely to be negatively contributing, i.e., "disturbing" the results. Figure 4 depicts the correlation between constructs and principal components in the form of a correlogram (Friendly 2002).

In Fig. 4, blue values represent positive correlations, while red values indicate negative correlations. Higher absolute values indicate stronger correlations. Thus, the closer to $+1$ the correlation is, the greater the contribution of the construct. Similarly, constructs with correlations close to -1 are more likely disturbing the effectiveness of the technique. The strength of the correlation is also given by the intensity of the color: dark blue and dark red circles indicate strong positive and negative correlations, respectively. Statistically insignificant correlations are signaled with a dark "$\times$".

Positive correlations. Figure 4 shows that many constructs are positively correlated with the first principal component (Dim.1). Not surprisingly, method and class names are the ones with the strongest positive correlation. This means that method and class names are the most influential constructs regarding the first dimension extracted by the PCA. This result was expected as classes and methods often represent the most important domain abstractions realized in program files. They embrace some other inner constructs in a file,

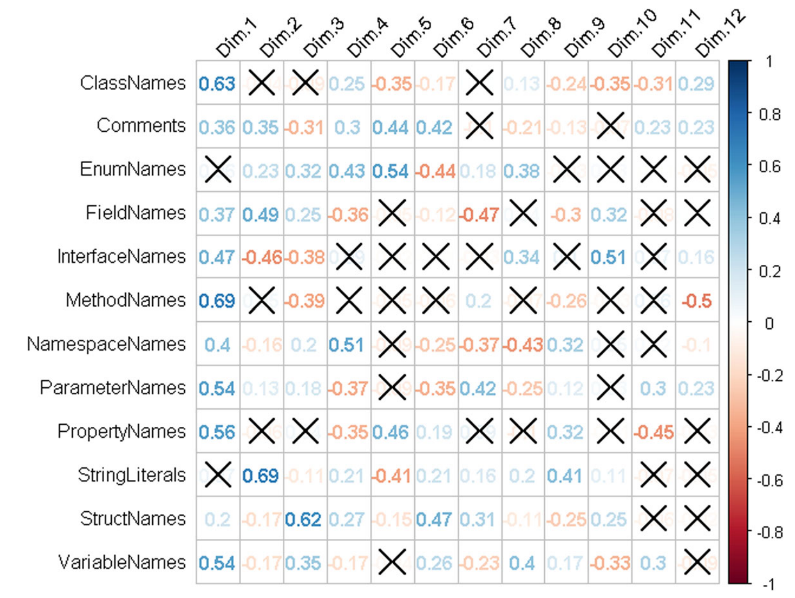

Fig. 4 Correlation between variables and principal components

were the bugs are often "located". Given their importance in the system domain, the names of such (class or method) abstractions are often the target of reflection when someone is either reporting or locating a bug.

The construct with the third highest correlation to the first PC is *Properties*. *Properties*, alongside with *Structures*, is one of the two C# constructs (Table 5) that have no equivalent in Java. The contribution of *Properties*, though, was more relevant than that of *Structures*. This can be explained by the fact that *Structures* usually represent simple data structures, with little or no behavior, and therefore are less prone to be associated with bugs. Moreover, *Structures* are independent constructs, while *Properties*, on the other hand, are members of classes. Therefore, it is expected that *Properties* be more closely related to domain abstractions, increasing their chances to be mentioned in bug reports.

After *Properties*, the next constructs more correlated with PC1 are *Parameters* and *Variables*. *Variables* represent a ubiquitous concept of programming languages, and its relevance to IR-based bug localization is no surprise. *Parameters* are used to pass values or variable references to methods (Microsoft Corporation 2012). Although *Parameters* are strictly different from *Variables*, their purposes are quite similar. We have discussed the possibility of considering some constructs equivalent, including *Parameters* and *Variables*, with the *Mixed* construct-mapping mode (Section 3.6). However, we have observed the *Complete* mode, i.e., considering the constructs separately, yielded better results (Section 3.7). The high correlation of these two constructs with PC1 may explain the advantage of the *Complete* mode. As both constructs proved to be relevant (Fig. 4), considering them separately had the effect of raising the similarity score.

The construct with the strongest positive correlation with the second dimension is *String literals*. This construct had a negligible effect on the first principal component. However, the strong correlation with the second component indicates that, overall, it still has a significant contribution to bug localization effectiveness. The importance of *String literals* may be explained by the fact that many bug reports include error messages, which are often included in the source code as string literals. This finding also reinforces that *String literals* should be explicitly considered in structured IR-based bug localization models.

Moreover, the fact that *String literals* were more correlated with the second PC, rather than the first, is meaningful. As aforementioned, each PC represents a different dimension of the data. Thus, the contribution of *String literals* occurs in a different dimension than that represented by PC1. This means that files with high scores due to similarity with *String literals* did not have high scores due to method or class name similarity, for example. This fact can be interpreted as an indication that some files would only be located due to the similarity of bug reports with *String literals*. This is an interesting result, as *String literals* were not considered by BLUiR (Saha et al. 2013) nor AmaLgam (Wang and Lo 2014), despite being a frequently used construct.

Similar reasoning can be applied to the third PC, where *Structures* are the most relevant construct, and so forth. However, as one advances into the subsequent PCs, one must remember that the relevance of the PCs decreases (Fig. 3). Moreover, constructs with negative correlations become more common. Thus, an analysis of the influence of negative correlations is also necessary.

Negative correlations. No construct showed negative correlation with the first principal component. However, from the second PC onwards, negative correlations start to appear. The highest negative correlations observed were for *Methods*, on PC12 (−0.5), followed by *Fields* on PC7 (−0.47), and *Interfaces* on PC2 (−0.46). However, the percentage of the variance explained by these components are 4.1%, 6.7%, and 10%, respectively (Fig. 3). Thus, the strong negative correlation displayed by *Interfaces* represent a more relevant concern.

Interfaces presented a strong negative correlation as early as the second dimension. Although it was also responsible for a similar contribution on PC1, its relative influence within that particular PC was lower than in PC2 and PC3: it has the sixth largest absolute correlation value on PC1 and the third largest value on PC2 and PC3. Apart from PC1, the positive contributions from *Interfaces* appear only on PC8 (fourth largest) and PC10 (first largest). These dimensions, however, account for 6.6% and 5.7% of the variance observed in the scores. Therefore, the positive contribution from *Interfaces* are relatively low, compared to other constructs. The negative correlation of *Interfaces* with PC2 suggests this construct offers the most negative contribution, thus probably decreasing bug localization effectiveness. This is somehow a surprising result since interface names usually capture important domain abstractions, which could frequently be mentioned in bug reports. However, we observed that in our dataset: (i) bug reports tend to refer to concrete concepts rather than abstract concepts (interfaces), and (ii) bugs are unlikely to be directly or indirectly related to the realization of an interface in the program.

4.2 Effects of constructs on bug localization results

This section explores the effects of program constructs on bug localization. Section 4.1.4 presented the constructs of interest for such exploration, i.e., constructs with low and high correlations with bugs effectively located by AmaLgam. To investigate the effects of these constructs, we have formulated the following questions:

RQ4: Does the effectiveness of bug localization increase with the suppression of constructs with the lowest contributions?

RQ5: Does the effectiveness of bug localization increase with the emphasis on constructs with the highest contributions?

RQ4 asks whether the suppression of low-contributing constructs could increase bug localization effectiveness. *Interfaces* emerged as the construct with higher negative correlation—thus, less correlated—with effectively located bugs. Therefore, RQ4 will be answered by adapting AmaLgam to ignore interface names and, then, applying this adapted version on the projects that comprise the experimental dataset.

In contrast, RQ5 inquires about the effects of emphasizing constructs highly correlated with bugs that were effectively located by AmaLgam, namely, *Methods* and *Classes* (Section 4.1.4). Both RQs can be answered by running AmaLgam's adaptation presented in the next section.

4.2.1 Modifying the emphasis of program constructs

To answer RQ4 and RQ5, we modify AmaLgam by allowing it to use different weights for each file part generated during source file splitting. The formula originally defined in BLUiR (Saha et al. 2013), and also used by BLUiR+ (Saha et al. 2013) and AmaLgam (Wang and Lo 2014) (Equation 1) is replaced by:

$$sim(f, b, w) = \frac{\sum_{i=1}^{n} \left[w_i \sum_{bp \in b} sim(fp_i, bp) \right]}{\sum_{i=1}^{n} w_i} \tag{4}$$

Equation 4 incorporates weights to the calculation performed by AmaLgam's structure component. Recall from Section 2.3.1 that structural similarity is computed by splitting bug reports and source files into parts corresponding to relevant fields. Bug reports are split into summary and description, while source files are split into as many parts as the number of constructs being used.

Using this fomula will allow us to modify the influence of constructs by attributing them weights. For instance, if *Methods* and *Classes* are the constructs that contribute the most to the technique, it is possible that attributing higher weights to their scores could lead to an effectiveness increase. Likewise, if *Interfaces* are negatively contributing to the result, effectiveness may be increased by nulling the similarity scores related to interface names. Next subsections presents the results of this evaluation.

4.2.2 Suppression of low-contributing constructs

The influence of each program construct on the similarity scores attributed by AmaLgam is not homogeneous (Section 4.1.4). The correlation of these scores with the dimensions revealed by principal component analysis (Fig. 4) made it clear that some constructs exert greater influence on bug localization effectiveness than other constructs do. It is unclear, however, whether negative correlations can disturb results. That is the subject of RQ4:

RQ4: Does the effectiveness of bug localization increase with the suppression of constructs with the lowest contributions?

Principal component analysis results showed that the constructs with the lowest contribution are *Interfaces*. In fact, *Interfaces* are the constructs with the larger negative correlation with the principal components. Thus, while similarity scores from positively correlated constructs increase together, scores from interface names decrease. To assess whether this effect has any influence in bug localization results, we used the dataset of 20 C# projects to run AmaLgam considering all the 12 available C# constructs (Table 5) except for *Interfaces*. Results are summarized in Table 8.

Removing *Interfaces* from the localization process increased AmaLgam's average MAP from 0.244 to 0.245 (0.4%). Median and maximum MAPs were also slightly increased, while minimum MAP was unchanged. This is a positive, although negligible, increase on AmaLgam results, with no statistical significance. Therefore, it is not possible to answer RQ4 positively based on our dataset, as it cannot be said that *Interfaces* hamper bug localization. Thus, contrary to our initial assumption, it is not needed to remove low-contributing constructs from bug localization models based on structured information retrieval.

4.2.3 Emphasis on most contributing constructs

One possible way of increasing effectiveness of bug localization based on structured information retrieval is to assign different weights to the parts in which source files are split (Saha et al. 2013). PCA revealed that *Methods* and *Classes* are the constructs with greater contribution to bug localization results (Section 4.1.4). Thus, RQ5 asks whether it is possible to increase the effectiveness of a technique by emphasizing highly contributing constructs:

RQ5: Does the effectiveness of bug localization increase with the emphasis on constructs with the highest contributions?

To answer RQ5, we must choose one or more constructs with high contributions to the results, assign them higher weights (Equation 4), and re-run AmaLgam with this configuration. We selected the two constructs with the highest contribution, *Methods* and *Classes* (Section 4.1.4), and assigned weights of 1.5, 2.0, and 3.0 to each one. These values were arbitrarily chosen to promote a significant variation in the weights, so we could observe to which extent the technique benefits from using higher or lower weights. The results obtained with this execution are displayed in Table 9. The first row repeats AmaLgam best results, i.e., with the *Complete* mode (Table 7), while next rows (referenced by keys) represent the weighted configurations being tested.

Table 9 shows that, in general, usage of higher weights was able to increase AmaLgam's effectiveness, measured in terms of mean average precision (MAP). *Class* constructs (rows D–F) led to higher MAPs than *Method* constructs (rows A–C) with the same weight for all statistics (minimum, median, maximum and average MAP). As for the weight values selected, best average MAPs were obtained when the emphasized construct had its weight doubled (rows B and E, weight = 2.0).

We used Wilcoxon Signed-Rank tests to assess statistical significance. Unfortunately, none of the results was statistically significant at 95% confidence level, although configurations with weights = 1.5 (rows A and D) came close (94.7% for *Method* and 90.8% for *Class*). The confidence levels decreased drastically as the weights increased. For instance, the result for the configuration with the highest MAP, i.e., *Class* weight = 2.0 (row E), had

Table 8 Effect of the suppression of interface names—MAP

Mode	Min	Median	Max	Avg.
All constructs	0.055	0.198	0.573	0.244
Without interfaces	0.055	0.200	0.582	0.245

Table 9 Effect of applying higher weights to method and class names—MAP

Key	Mode	Min	Median	Max	Avg.
—	*Baseline*	*0.055*	*0.198*	*0.573*	*0.244*
A	Method weight = 1.5	0.055	0.200	0.574	0.246
B	Method weight = 2.0	0.056	0.195	0.574	0.246
C	Method weight = 3.0	0.050	0.171	0.574	0.238
D	Class weight = 1.5	0.055	0.207	0.582	0.248
E	Class weight = 2.0	0.055	0.207	0.582	0.265
F	Class weight = 3.0	0.055	0.194	0.582	0.256

a confidence level of 73% (p-value = 0.2707). As for *Class* weight = 3.0 (row F), not only the MAP dropped, but also the confidence level (32%, p-value = 0.6783). The same was observed for *Method* weight = 3.0 (row C), which means 3.0 is a weight value beyond the threshold both for effectiveness and for significance.

The constructs *Methods* and *Classes* presented similar levels of influence to bug localization results, as measured by their correlation to the main component revealed by PCA analysis (Fig. 4). Thus, we also tested AmaLgam simultaneously changing the weights of these two constructs. We fixed *Class* weight with a value of 2.0, as it was the best result obtained when constructs had their weights changed individually (Table 9, row E). Then, we applied weights of 1.5 and 2.0 to *Method* constructs. We did not set *Method* weight = 3.0, as this weight value led to smaller MAPs for both constructs evaluated individually (Table 9, rows C and F). Results are presented in Table 10.

In Table 10, previous results (in *italics*) are repeated for the sake of comparison. The first row contains baseline results from the *Complete* mode, with equal weights for all constructs (Table 7). The second row repeats the result obtained with a weight of 2.0 attributed to *Class* constructs (Table 9, row E). It is possible to see that average MAP increased from 0.244 (baseline) to 0.266 when *Method* weight is set to 1.5 (row G), and to 0.253 when *Method* weight is 2.0 (row H). Combining *Method* weight = 1.5 and *Class* weight = 2.0 (row G) even increased average MAP compared to using only *Class* weight = 2.0 (row E), although by a negligible amount (only 0.4% higher, from 0.265 to 0.266).

As with the first part of this evaluation, we used Wilcoxon Signed-Rank tests to determine statistical significance. Results for combined weights were closer to the selected 95% confidence threshold: 93% for row G and 96% for row H (p-values of 0.0682 and 0.0412, respectively). Thus, it is possible to state that setting *Class* and *Method* weights to 2.0 (row H) significantly increased bug localization effectiveness, compared with the baseline with equal weights for all constructs.

Table 10 Effect of combining higher weights on method and class names—MAP

Key	Mode	Min	Median	Max	Avg.
—	*Baseline*	*0.055*	*0.198*	*0.573*	*0.244*
E	*Class weight = 2.0*	*0.055*	*0.207*	*0.582*	*0.265*
G	Class weight = 2.0 Method weight = 1.5	0.055	0.198	0.571	0.266
H	Class weight = 2.0 Method weight = 2.0	0.056	0.195	0.571	0.253

4.2.4 Discussion

In this section, we investigated the influence of different program constructs on the effectiveness of structured IR-based bug localization. Initially, we applied principal component analysis (PCA) on results from AmaLgam in the *Complete* mode. This analysis intended to reveal which constructs from the C# language exerted more or less influence on bug localization results.

PCA data suggested that all constructs exerted a significant level of influence on the results (Section 4.1.3). Thus, it was not possible to identify irrelevant constructs just by inspecting PCA data. The analysis also revealed that *Methods* and *Classes* were the constructs with more influence on the results (Section 4.1.4).

In spite of PCA data not having revealed constructs that could be considered irrelevant, some constructs emerged as negatively correlated with bug localization results. The most striking example was *Interfaces* (Fig. 4). This negative correlation caused us to investigate what would be the effect of suppressing *Interfaces* from bug localization (Section 4.2.2). Compared to the baseline, results were practically unchanged (Table 8). Thus, we conclude that suppression of low-contributing constructs does not increase bug localization effectiveness.

Another possible way of increasing effectiveness of bug localization is by emphasizing constructs that are more influential, i.e., *Methods* and *Classes* (Section 4.1.4). Although this possibility has been suggested in the literature (Saha et al. 2013), we were unable to find any bug localization technique that emphasize particular program constructs. We investigated that possibility by running AmaLgam with alternative configurations, where different weights were assigned to these two constructs, one at a time (Section 4.2.3). Practically all of these configurations caused the average MAP to increase (Table 9), although none of these improvements reached our statistical significance threshold. Nonetheless, we also tested AmaLgam assigning higher weights to both *Methods* and *Classes*, simultaneously. In this case, a statistically significant improvement was attained when *Methods* and *Classes* were assigned a weight of 2.0 (Table 10). Compared to the baseline, MAP increased from 0.244 to 0.253 (3.7%).

It was previously demonstrated that bug localization based on structured information retrieval benefits from the usage of more program constructs (RQ2, Section 3.7). This finding is reinforced by the thorough analysis of the contribution of program constructs performed in this section. The answer to RQ3 suggested that all constructs significantly influence bug localization results. RQ4 confirmed this suspicion, by showing there was no significant effectiveness increase when the technique ignored the construct with the smallest contribution.

The usage of weights in the calculation of structural similarity increased bug localization effectiveness. The weight values used in this experiment (1.5, 2.0, and 3.0) were selected empirically. Thus, a possible improvement to this evaluation involves determining optimal weights for each construct. Likewise, we only evaluated the assignment of higher weights to the two most influential constructs, i.e., *Methods* and *Classes*. However, the effect of weighing more than two constructs is still unknown, and could be the subject of future studies. Nonetheless, weighing constructs proved to be a promising way of increasing the effectiveness of bug localization techniques based on structured information retrieval.

5 Conclusion

Structured information retrieval has been successfully applied to the bug localization problem. Techniques based on structured IR have shown to be considerably more effective than other IR-based approaches. However, these techniques are language-specific, as they depend upon the structure of source files. Considering the multi-language nature of most modern software (Karus and Gall 2011), it is important to have effective bug localization models for the different kinds of languages and technologies used in software projects. This study is a step in that direction, where a thorough evaluation is performed on C# projects, offering an alternative to the usual choice for Java projects.

Next sections summarize our findings and present ideas for future work.

5.1 Findings

The average effectiveness of the evaluated techniques on C# projects was lower than the same metric reported in the original studies on Java. However, some projects have individually yielded results above the average informed by the Java studies. Therefore, we conclude that, in principle, there is no impediment to the usage of bug localization on C# projects due to features of the language itself. The lower average effectiveness compared to previous studies can be attributed to (i) the 5x higher number of projects evaluated, and (ii) the discard of localized bug reports, which artificially increased the effectiveness of bug localization techniques in previous studies. We also demonstrated that using more program constructs, which is a strategy that differs from previous studies (Saha et al. 2013; Wang and Lo 2014), increased bug localization effectiveness by 18% on average.

As the usage of more constructs was shown to increase bug localization effectiveness, it became important to study the individual contribution of each construct type. Using Principal Component Analysis, we have shown that all constructs exerted a significant level of influence on the results, with *Methods* and *Classes* being the most influential construct types (Section 4.1.4). *Properties*, *Parameters*, *Variables*, and *String literals* also had a significant impact on the results. The fact that *String literals* are among the most contributing constructs is of particular interest, since this construct was not considered by previous techniques (Saha et al. 2013; Wang and Lo 2014), in spite of its ubiquitous presence among programming languages. This finding suggests that existing techniques could have achieved better results if *String literals* had been explicitly considered.

The fact that *Interfaces* offered the most negative contribution (Section 4.1.4) is also surprising. Since *Interfaces* often capture key domain abstractions, it would be expected that these constructs had a positive impact on bug localization effectiveness. However, in spite of PCA data suggesting that *Interfaces* actually *disturbs* the results (Section 4.1.4), we could not confirm that in practice. When AmaLgam was adapted to consider all the 12 C# constructs (Table 5) except *Interfaces*, MAP has, in fact, increased (Table 8). However, the improvement was negligible, and not statistically significant. Therefore, we conclude there is little gain in removing constructs from bug localization techniques based in structured IR.

The effect of the most contributing constructs was answered by running AmaLgam with different weights assigned to each program construct. We assigned various weight values to *Method* and *Class* constructs, both in isolation (Table 9) and combined (Table 10). Most of the values tested increased bug localization effectiveness. However, a statistically significant improvement was achieved with weight values of 2.0 for both *Classes* and

Methods. The weight values were empirically determined, suggesting there might be optimal values that lead to even better results. Nonetheless, this result shows that structured IR-based techniques can be fine-tuned to increase effectiveness even further.

5.2 Future work

Future work includes the creation of a standard bug dataset containing bugs from C# projects. This would allow studies with better potential for generalizability involving the C# language. It is also important to validate bug localization techniques with user experiments, as pointed by Wang et al. (2015). Meanwhile, analytical studies to improve bug localization knowledge on different languages and different types of files could be carried out. For instance, defects are not always located in source files. Sometimes they can be found in different kinds of files, like configuration files. Developers could benefit from having specific localization techniques for these kinds of bugs.

Weighted similarity calculation was shown to increase bug localization effectiveness (Section 4.2.3). However, our evaluation selected weight values empirically. The effectiveness could be further increased if an optimal set of weights could be found. The calculation of the weights could be automated and performed on a per project basis, which might lead to even better results.

Finally, our study implements the experimental steps needed to solve the issue of different versions raised by Rao and Kak (2013a) and the *localized bug report* bias presented by Kochhar et al. (2014). Besides, we also observed that test files should not be included in the scope of the localization process. These steps are important because they demonstrate that the reported effectiveness of current state-of-the-art bug localization techniques cannot be achieved in realistic situations. Future studies in bug localization should not skip such steps, as they produce an experimental setup closer to reality and to developers' expectations, increasing the chances of bug localization to become more useful in practice.

Endnotes

[1] AspectJ, Eclipse, SWT and ZXing.

[2] https://github.com/search/advanced

[3] GitHub API limits issue searching to 1000 results per query.

[4] All the distributions obtained in the study deviate from normality according to the Shapiro-Wilk test.

Funding
The authors acknowledge financial support in this research from Petróleo Brasileiro S. A - Petrobras. This work is also funded by CAPES/Procad (grant #175956), CNPq (grants #483425/2013-3 and 309884/2012-8), and FAPERJ (E26-102.166/2013).

Availability of data and materials
An online appendix for this study is available (Garnier 2016). The appendix includes the performed statistical analysis and a study replication package.

Authors' contributions
MG helped to conceive and design the study, conducted the data collection and analysis, and drafted the manuscript. IF participated in the data analysis and helped to draft the manuscript. AG helped to conceive and design the study, coordinated the research activities and helped to draft and review the manuscript. All authors read and approved the final manuscript.

Competing interests
The authors declare that they have no competing interests.

Publisher's Note

Received: 16 December 2016 Accepted: 30 August 2017
Published online: 08 September 2017

References

Bachmann A, Bernstein A (2009) Data retrieval, processing and linking for software process data analysis. Technical Report IFI-2009.0003b, Department of Informatics (IFI), University of Zurich. http://www.merlin.uzh.ch/publication/show/2525

Baeza-Yates R, Ribeiro-Neto B (1999) Modern Information Retrieval. ACM Press, New York

Dallmeier V, Zimmermann T (2016) iBUGS. https://www.st.cs.uni-saarland.de/ibugs/. Accessed Nov 2016

Friendly M (2002) Corrgrams: Exploratory displays for correlation matrices. Am Stat 56(4):316–324

Garnier M (2016) Bug localization in C#. https://mgarnier.github.io/bug_localization. Accessed Nov 2016

Garnier M, Garcia A (2016) On the evaluation of structured information retrieval-based bug localization on 20 C# projects. In: Proceedings of the 30th Brazilian Symposium on Software Engineering. SBES '16. ACM, New York. pp 123–132. doi:10.1145/2973839.2973853. http://doi.acm.org/10.1145/2973839.2973853

Jolliffe IT (2002) Principal Component Analysis. Springer, Secaucus

Karus S, Gall H (2011) A study of language usage evolution in open source software. In: 8th Working Conference on Mining Software Repositories (MSR). MSR '11. ACM, New York. pp 13–22. doi:10.1145/1985441.1985447. http://doi.acm.org/10.1145/1985441.1985447

Kochhar PS, Tian Y, Lo D (2014) Potential biases in bug localization: Do they matter? In: 29th International Conference on Automated Software Engineering (ASE). pp 803–814. doi:10.1145/2642937.2642997. http://doi.acm.org/10.1145/2642937.2642997

Lewis C, Ou R (2011) Bug Prediction at Google. http://google-engtools.blogspot.sg/2011/12/bug-prediction-at-google.html. Accessed Nov 2016

Lukins SK, Kraft NA, Etzkorn LH (2010) Bug localization using latent dirichlet allocation. Inf Softw Technol 52(9):972–990

Manning CD, Raghavan P, Schütze H (2008) Introduction to Information Retrieval. Cambridge University Press, Cambridge

Microsoft (2014) NET Compiler Platform ("Roslyn"). https://github.com/dotnet/roslyn

Microsoft Corporation (2012) C# Language Specification 5.0. https://www.microsoft.com/download/details.aspx?id=7029. Accessed July 2016

Rahman F, Posnett D, Hindle A, Barr E, Devanbu P (2011) Bugcache for inspections: Hit or miss? In: 19th ACM SIGSOFT Symposium and the 13th European Conference on Foundations of Software Engineering. pp 322–331. doi:10.1145/2025113.2025157. http://doi.acm.org/10.1145/2025113.2025157

Rao S, Kak A (2011) Retrieval from software libraries for bug localization: A comparative study of generic and composite text models. In: 8th Working Conference on Mining Software Repositories (MSR). pp 43–52. doi:10.1145/1985441.1985451. http://doi.acm.org/10.1145/1985441.1985451

Rao S, Kak A (2013a) moreBugs. https://engineering.purdue.edu/RVL/Database/moreBugs/#C5. Accessed Nov 2016

Rao S, Kak A (2013b) moreBugs: A new dataset for benchmarking algorithms for information retrieval from software repositories. Technical Report TR-ECE-13-07. http://docs.lib.purdue.edu/ecetr/447

Saha RK, Leasey M, Khurshid S, Perry DE (2013) Improving bug localization using structured information retrieval. In: 28th International Conference on Automated Software Engineering (ASE). pp 345–355. doi:10.1109/ASE.2013.6693093

Sisman B, Kak AC (2012) Incorporating version histories in information retrieval based bug localization. In: 9th Working Conference on Mining Software Repositories (MSR). pp 50–59

The GitHub Blog (2015) Language Trends on GitHub. https://github.com/blog/2047-language-trends-on-github. Accessed Nov 2016

TIOBE Software BV (2016) TIOBE Index for April 2016. http://www.tiobe.com/tiobe_index. Accessed April 2016

Wang Q, Parnin C, Orso A (2015) Evaluating the usefulness of ir-based fault localization techniques. In: 2015 International Symposium on Software Testing and Analysis (ISSTA). ISSTA 2015. ACM, New York. pp 1–11. doi:10.1145/2771783.2771797. http://doi.acm.org/10.1145/2771783.2771797

Wang S, Lo D (2014) Version history, similar report, and structure: Putting them together for improved bug localization. In: ACM (ed). 22nd International Conference on Program Comprehension (ICPC). pp 53–63. doi:10.1145/2597008.2597148. http://doi.acm.org/10.1145/2597008.2597148

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: 34th International Conference on Software Engineering (ICSE). pp 14–24