Boosting fault localization of statements by combining topic modeling and Ochiai

Romain Vacheret, Francisca Pérez, Tewfik Ziadi, Lom Hillah

Boosting Fault Localization of Statements by Combining Topic Modeling and Ochiai

Romain Vacheret[a,*], Francisca Pérez[b], Tewfik Ziadi[a], Lom Hillah[c]

[a]*Sorbonne Université CNRS, LIP6,*
Campus Pierre et Marie Curie 4 place Jussieu 75005 Paris, France
[b]*Universidad San Jorge. SVIT Research Group*
Autovía A-23 Zaragoza-Huesca Km.299, 50830, Zaragoza, Spain
[c]*NewCo Partners,*
La Grande Arche – Paroi Nord 1, parvis de la défense 92044 Paris La Défense cedex, France

Abstract

Context: Reducing the cost of maintenance tasks by fixing bugs automatically is the cornerstone of Automated Program Repair (APR). To do this, automated Fault Localization (FL) is essential. Two families of FL techniques are Spectrum-based Fault Localization (SBFL) and Information Retrieval Fault Localization (IRFL). In SBFL, the coverage information and execution results of test cases are utilized. Ochiai is one of the most effective and used SBFL strategies. In IRFL, the bug report information is utilized as well as the identifier names and comments in source code files. Latent Dirichlet Allocation (LDA) is a generative statistical model and one of the most popular topic modeling methods. However, LDA has been used at the method level of granularity as IRFL technique, whereas most existing APR tools are focused on the statement level.

Objective: This paper presents our approach that combines topic modeling and Ochiai to boost FL at the statement level.

Method: We evaluate our approach considering five different projects in Defects4J benchmark. We report the performance of our approach in terms of hit@k and MRR. To study the impact on the results, we compare our approach against five baselines: two SBFL approaches (Ochiai and Dstar), two IRFL approaches (LDA and Blues), and one hybrid approach (SBIR). In addition, we compare the number of bugs that are found by our approach with the baselines.

Results: Our approach significantly outperforms the baselines in all metrics. Especially, when hit@1, hit@3 and hit@5 are compared. Also, our approach locates more bugs than Ochiai and Blues.

Conclusion: The results of our approach indicate that the integration of topic modeling with Ochiai boosts FL. This uncovers the potential of topic modeling for FL at statement level, which is valuable for the APR community.

Keywords: fault localization, topic modeling, information retrieval fault localization, spectrum-based fault localization

[*]Corresponding author.
Email addresses: romain.vacheret@lip6.fr (Romain Vacheret), mfperez@usj.es (Francisca Pérez), tewfik.ziadi@lip6.fr (Tewfik Ziadi),

1. Introduction

Automated Program Repair (APR) aims to reduce the cost of fixing bugs by automatically producing patches. In APR, it is essential automated Fault Localization (FL) to identify the suspicious program statements that may be the cause of the failure. Recent studies show that accuracy of the FL used by APR has a significant effect on the success of APR [1]. Two existing families of FL techniques utilize either test coverage information (Spectrum-based Fault Localization) or the bug report information (Information Retrieval Fault Localization).

On the one hand, Spectrum-based Fault Localization (SBFL) is very frequently used for APR [2]. SBFL techniques typically analyze program spectra that corresponds to program elements, which are executed by failing and successful execution traces. As a result, a ranked list of program elements (typically program blocks or statements) is obtained. The ranking takes into account the program elements that are executed more often in the failing rather than correct traces. In SBFL, GZoltar with the Ochiai ranking strategy is one of the most effective ranking strategies in object-oriented programs [3] and most APR tools use it [4, 5, 1, 6].

On the other hand, Information Retrieval Fault Localization (IRFL) typically analyzes textual descriptions contained in bug reports, and identifier names (variables, classes...) and comments in source code files. IRFL approaches take as input a bug report and generate as output a ranked list of suspicious program elements (typically files or methods) [7]. A popular IRFL technique is Latent Semantic Indexing (LSI) [8], which analyzes relationships between queries (e.g., bug report) and documents (e.g., source code files).

Another technique that has previously shown promising results in a variety of retrieval tasks in source code [9, 10] and in software models [11] is Latent Dirichlet Allocation (LDA) [12]. LDA is one of the most popular topic modeling methods [13]. LDA is a generative statistical model that has significant advantages, in modularity and extensibility, over LSI [14]. As IRFL technique, LDA has been used at the method level of granularity [14]. However, most APR tools [5, 1, 6] work at the statement level of granularity. If a given bug report shares topic(s) with a code statement, the code statement could be a strong candidate towards causing the failure. In this context, we theorize that the coarse-grained focus (topics) of LDA could enhance IRFL.

In this paper, we propose to combine topic modeling (LDA) and Ochiai to boost fault localization at the statement level of granularity. Thus, APR tools can integrate our approach for FL. Although previous works improve localization by combining multiple FL techniques [15, 1], to the best of our knowledge, this work is the first effort that investigates fault localization of statements by combining topic modeling and Ochiai in the literature.

For the evaluation, we use Defects4J, which is a well-known Java benchmark that has been used in previous FL and APR works [5, 1, 6]. The inputs from Defects4J are bug reports, test cases and source code from a total of 59 bugs of 5 different projects. The output of our approach is a ranking of suspicious statements from the provided inputs.

We assess the performance of our approach by considering two combination strategies (i.e., variants) to obtain the ranking of suspicious statements. The first variant obtains the mean suspiciousness score between the obtained score of topic modeling and Ochiai. The second variant considers the position of the statement in the IRFL and SBFL ranking using a strategy that is inspired by the Borda Count family [16]. This strategy applies more weight to the values of topic modeling and Ochiai as the position in the ranking is better.

To put the performance of our approach in perspective and to study the impact on the results, we compare the results of the variants of our approach against five baselines. Baseline$_1$ is a SBFL approach that uses GZoltar with the Ochiai ranking strategy (most APR tools use it [4, 5, 1, 6] and it is effective in object-oriented programs [3]). Baseline$_2$ is also a SBFL approach that uses GZoltar but with the DStar ranking strategy (another popular ranking strategy [17]). Baseline$_3$ is a IRFL approach

lom.hillah@newco-partners.com (Lom Hillah)

that uses LDA as the topic modeling technique. Although the application of LDA for FL in statements is novel, we set it as baseline to compare its performance with our approach, which combines the results of topic modeling and Ochiai. Baseline$_4$ is Blues, which is a recent statement-level IRFL approach [1] that uses an IR model (Term Frequency-Inverse Document Frequency formulation) to search and rank the statements based on their similarity with the bug report. Baseline$_5$ is SBIR, which is a recent statement-level approach [1] that combines the results of Blues (Baseline$_4$) with SBFL (GZoltar with the Ochiai strategy as in Baseline$_1$).

We report the performance of our approach and the baselines in terms of solution quality using metrics that are employed in previous FL studies [18, 19, 20, 21, 22, 1, 23, 24, 17, 25]: the commonly used hit@k and Mean Reciprocal Rank (MRR). Hit@k is the number of bugs localized in the top-k ranked statements, whereas MRR measures the quality of the ranking of the FL technique by capturing how close to the top of ranking a target (i.e., faulty) statement is retrieved. In addition, we perform a statistical analysis that includes statistical significance (Holm's post-hoc) and a effect size measurement (Vargha and Delaney's $\hat{A}_{12}$ [26]) to show whether the impact of our approach is significant and if so, by how much (following the guidelines by Arcuri and Briand [27]). Furthermore, we compare the number of bugs that are found by our approach and the baselines in order to determine whether our approach locates bugs that go undetected by the baselines (and vice versa).

The results show that the two variants of our approach significantly outperform the baselines in all metrics by a large effect size. The highest differences between our approach and the baselines are when hit@1, hit@3 and hit@5 are compared. Using the two variants of our approach, 20.34% of the total number of bugs are ranked in the hit@1, compared to 8.85% for Baseline$_3$ (LDA), 6.77% for Baseline$_1$ (Ochiai) and Baseline$_5$ (SBIR); and 3.38% for Baseline$_4$ (Blues). Also, our approach not only locates all bugs that are detected by the baselines, but also locates 5 bugs that go undetected by Baseline$_1$ (Ochiai), and 7 of the bugs that go undetected by Baseline$_4$ (Blues).

The main contributions of this paper are listed as follows:

- We investigate the potential of combining topic modeling and Ochiai for fault localization of statements.
- We particularly assess the performance of our approach in Defects4J, which is used in previous FL and APR works [5, 1, 6].
- We compare the results of our approach against five baselines. Thus, the performance of our approach is put in perspective.
- We provide an online replication package[1] that includes: the code of our approach that enables its usage in other FL and APR works, the data that is used as input in the evaluation, and the data resulting from our evaluation that enable the reproducibility of the results.

The remainder of the paper is structured as follows. The preliminary knowledge related to SBFL, IRFL and topic modeling is reviewed in Section 2. Section 3 presents our approach and the two variants to combine SBFL and IRFL. Section 4 presents our evaluation in Defects4J. Results are reported in Section 5 and discussed in Section 6. Section 7 discusses the threats to validity that could have affected our evaluation. Section 8 presents the related works. Finally, Section 9 concludes the paper.

2. Background

Fault localization plays an important role in software development and debugging processes. The primary objective of fault localization is to pinpoint the exact locations or sections of code responsible for software failures or anomalies. Various techniques and approaches have been proposed to tackle the challenge of fault localization, including Spectrum-Based Fault Localization (SBFL), Information Retrieval-based Fault Localization (IRFL). The goal of this paper is to enhance fault localization

[1] https://figshare.com/s/99831d5178d72cda36e1

by integrating existing techniques with topic modeling. To provide context for our approach, this section offers a concise background on the two primary fault localization families (SBFL and IRFL) and introduces the fundamental principles of topic modeling.

2.1. SBFL (Spectrum-Based Fault Localization)

SBFL aims to reveal abnormal execution patterns and thus detect potential bug locations. It is based on the execution of developer written tests cases and more specifically on the number of passing and failing tests. Therefore, it requires the program to have a test suite (i.e. a collection of test cases) and at least one failing test. The technique is based on the assumption that a faulty code entity is more likely to be present in more failing tests than in a correct one.

The code is instrumented to execute the available tests and extract execution information. Such information includes, for instance, the frequency of execution of the different code entities. This is called the program spectrum. There are several possible granularities for the code entities, such as file, method and statement. They are the same for the different fault localization approaches. Note that most techniques use either file or method granularity.

Once the test suite has been executed and the code entities frequency collected, the program spectrum is used to generate the suspiciousness scores. These scores serve as indicators of which code entities are more likely to be faulty and are calculated using a strategy.

One of the most well-known and popular fault localization techniques is Ochiai, as documented by Abreu et al. [28]. Ochiai has demonstrated notable effectiveness in analyzing object-oriented languages, as highlighted in the studies conducted by Xuan et al. [3] and Yang et al. [29]. The formula for Ochiai is presented as follows:

$$S_{ochiai}(e) = \frac{failed(e)}{\sqrt{(failed(e)+passed(e)) \times (failed(e)+failed(\neg e))}}$$

For each element e with a test coverage (i.e. at least one test executes it), three pieces of information are taken into account. The number of failing tests executing the element ($failed(e)$), the number passing tests executing the element ($passed(e)$) and the number of failing tests that do not execute the element ($failed(\neg e)$). The higher the score, the more likely the element is to be faulty. There are various other existing strategies, such as Jacard [28], DStar [17] or Tarantula [30], which are also used in many studies [3, 31, 32, 15, 24, 33].

JaCoCo [34] and Cobertura [35] are some of the tools used in the literature to compute code coverage. Particularly GZoltar [2] is the main SBFL tool using some of the most popular formulas.

2.2. IRFL (Information Retrieval Fault Localization)

Contrarily to SBFL which leverages test suites, IRFL uses bug reports written in natural language (see Figure 1). They are texts written in natural language, by users or developers, in order to track and describe an issue they have encountered. IRFL techniques are designed to find term relations between the bug report and the source code. The more terms are shared, the more likely the code entity (i.e. granularity such as file, method, statement) is to be faulty.

Issue: Cli-193 (Defects4J Cli-32)

Title: StringIndexOutOfBoundsException in HelpFormatter.findWrapPos

Description: In the last while loop in HelpFormatter.findWrapPos, it can pass text.length() to text.charAt(int), which throws a StringIndexOutOfBoundsException. The first expression in that while loop condition should use a <, not a <=.
...

Figure 1: Bug report example

Typically, the first step of such a technique is some text processing common to Natural Language Processing (NLP) tasks. Second, the algorithm is used to compute the similarity between bug report and the source code for a given granularity. Lastly, these similarities are used to rank the items and thus create a ranking. Along with SBFL, the rankings are sorted in descending order with higher scores meaning a higher probability for the element to be faulty.

Many IR algorithms have been used for fault localization, including Vector Space Model (VSM), LSI (Latent Semantic Indexing) and LDA (Latent Dirichlet Allocation) [36, 32, 37, 8]. VSM leverages both queries and documents as vectors of term weights. Both LSI and LDA are topic-based, meaning that they represent the correlation between terms contained in the documents and topics. To identify the term-document relationship, LSI uses Singular Value Decomposition (SVD) whereas LDA estimates topic-term and document-topic distributions using a probabilistic techniques as explained in Section 2.3. Note that LSI is also referred to as Latent Semantic Analysis (LSA).

Some of the techniques are supervised and some are not. Unsupervised models are easier to use than their counterparts, as they do not require training data, which is difficult and time-consuming to collect.

Almost all existing IR techniques have a file or method granularity. However, this is not suitable for APR, since it requires a ranking of statements as input.

2.3. Topic Modeling

Topic modeling aims to reduce a large text corpus into a set of meaningful topics. Each topic consists of a group of terms, collectively representing a potential thematic concept that pervades the collection. The power of topic modeling has motivated researchers to cover different software engineering activities including fault localization [14] and automated traceability [38].

One of the most popular topic modeling methods [13] for information retrieval is Latent Dirichlet Allocation (LDA) [12]. LDA is an unsupervised probabilistic technique for estimating a topic distribution over a text corpus. The corpus is made up of a set of documents, and each document contains a set of terms. As a result, each document receives a probability distribution, which indicates the likelihood that the document expresses each topic.

The main LDA inputs are the documents (D) and the number of topics (k) to be extracted. It is also necessary to set hyper-parameters, which have a smoothing effect on the topic model generated as output. The hyper-parameters of any LDA implementation are:

- k indicates the number of topics to be extracted from the documents.
- α controls the document-topic density. A lower α value means that there are fewer topics per document.
- β controls the topic-term density. A lower β value means that there are few words per topic, which in turn implies an increase in the number of topics needed to describe a particular document.

The LDA outputs are: ϕ, which is a matrix that contains the topic-term probability distribution, and θ, which is a matrix that contains the document-topic probability distribution.

Topic modeling as an IRFL technique has been used in previous works to rank files or methods [14, 39]. However, existing APR tools [4, 5, 1, 6, 40] require a ranking of statements as input.

3. Our approach

Most APR tools work at the statement level of granularity and use GZoltar and the Ochiai strategy as a SBFL technique [5, 1, 6], which relies on test suites. However, when using GZoltar some potential bugs go undetected (e.g., no tests pass through the statement or they do not fail). In this context, bug reports hold important information that can be exploited to improve the results of GZoltar. To address this, our approach OTM combines Ochiai and Topic Modeling at the statement level of granularity.

Figure 2 provides an overview of our approach. The top part of the figure highlights the inputs of the approach (the bug report, the source code, and the test cases), the middle part shows the two steps of our approach (fault localization and score combination), and the bottom part shows the output of the approach (a ranking of suspicious statements that might be the cause of the bug).

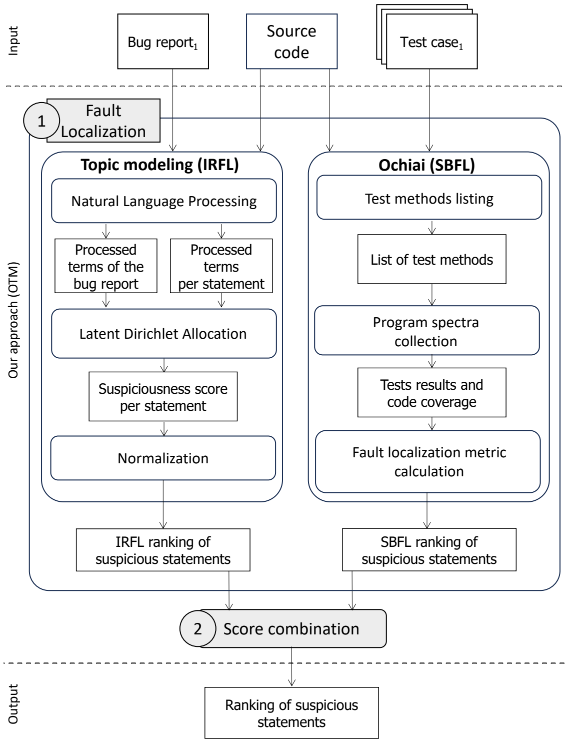
Input
Bug report1
Source code
Test case1
Our approach (OTM)
1
Fault Localization
Topic modeling (IRFL)
Natural Language Processing
Processed terms of the bug report
Processed terms per statement
Latent Dirichlet Allocation
Suspiciousness score per statement
Normalization
Ochiai (SBFL)
Test methods listing
List of test methods
Program spectra collection
Tests results and code coverage
Fault localization metric calculation
IRFL ranking of suspicious statements
SBFL ranking of suspicious statements
2
Score combination
Output
Ranking of suspicious statements

Figure 2: Approach overview

In the first step of our approach, the fault localization is performed with topic modeling (as the IRFL technique, taking the source code and the bug report as input) and Ochiai (as the SBFL technique, described in Section 2.1, taking the source code and the test cases as input). Once topic modeling and Ochiai are executed, an IRFL ranking and a SBFL ranking, respectively, are obtained.

In the next two sections, we present how we utilize topic modeling for FL in statements, and how we combine the IRFL and the SBFL rankings in the second step of our approach.

3.1. Topic modeling (LDA) for FL of statements

The middle-left part of Figure 2 shows the steps to obtain the IRFL ranking of suspicious statements using LDA as the topic modeling technique. First, the bug report and the source code are processed by means of NLP techniques as is usual in IRFL studies [1, 32, 39]. The text is lowercased, tokenized, and stemmed. In order to exclude the words commonly used in English, which would not add any value, the words are filtered using a stopword list. We also exclude the Java keywords to improve the performance. As a result, the processed terms of the bug report as well as the processed terms per statement of the source code are obtained. Figure 3 shows an example of the processed terms per statement and the processed terms of a bug report after the NLP techniques are applied.

Afterwards, we use LDA as the topic modeling technique to obtain the suspiciousness score per statement with regard to the bug report. Given the processed terms for the bug report as query (Q), each statement as document (D), and the outputs of LDA (ϕ and θ) as described in Section 2.3, the conditional probability P of Q given a document D_i is computed as follows [41]:

$$Sim(Q, D_i) = P(Q|D_i) = \prod_{q_k \in Q} P(q_k|D_i)$$

where q_k is the k^{th} homogenized term in the query (i.e., bug report), and D_i is a document (i.e., code statement) that is made of a set of homogenized terms.

Figure 3 shows an example of the process for assessing the suspiciousness score of the n statements using LDA as the topic modeling technique. The left part of the figure represents the outputs of LDA, which include ϕ and θ as described in Section 2.3. ϕ contains the term (K) to topic (T) probability distribution. Each cell can have a value from 0 to 1, indicating the likelihood of a term from the corpus being assigned to a particular topic. For instance, a value of $\phi[T_1, text.length] = 0.14$ indicates a 14% likelihood of the term $text.length$ being assigned to topic T_1. θ contains the topic (T) to statement (S) probability distribution. Each cell, with values that again range from 0 to 1, indicates the likelihood with which a statement expresses a topic. For instance, a value of $\theta[S_1, T_1] = 0.39$ indicates that statement S_1 has a 39% likelihood of expressing topic T_1.

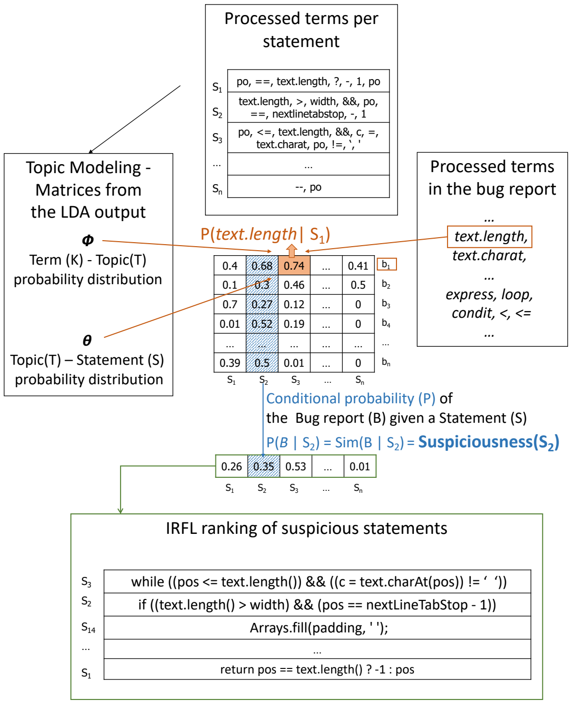
Processed terms per statement
S₁ po, ==, text.length, ?, -, 1, po
S₂ text.length, >, width, &&, po, ==, nextlinetabstop, -, 1
S₃ po, <=, text.length, &&, c, =, text.charat, po, !=, ', '
... ...
Sₙ --, po
Topic Modeling - Matrices from the LDA output
ϕ
Term (K) - Topic(T) probability distribution
θ
Topic(T) – Statement (S) probability distribution
Processed terms in the bug report
...
text.length,
text.charat,
...
express, loop, condit, <, <=
...
P(text.length | S₁)
0.4 0.68 0.74 ... 0.41 b₁
0.1 0.3 0.46 ... 0.5 b₂
0.7 0.27 0.12 ... 0 b₃
0.01 0.52 0.19 ... 0 b₄
... ...
0.39 0.5 0.01 ... 0 bₙ
S₁ S₂ S₃ ... Sₙ
Conditional probability (P) of the Bug report (B) given a Statement (S)
P(B | S₂) = Sim(B | S₂) = Suspiciousness(S₂)
0.26 0.35 0.53 ... 0.01
S₁ S₂ S₃ ... Sₙ
IRFL ranking of suspicious statements
S₃ while ((pos <= text.length()) && ((c = text.charAt(pos)) != ' '))
S₂ if ((text.length() > width) && (pos == nextLineTabStop - 1))
S₁₄ Arrays.fill(padding, ' ');
... ...
S₁ return pos == text.length() ? -1 : pos

Figure 3: Example of assessing suspicious statements using topic modeling

The LDA outputs (ϕ and θ) and the processed terms of the bug report are used to calculate a matrix with dimensions K x n. The rows of the matrix represent the processed terms of the bug report (b_n) and the columns of the matrix represent each of the statements (S_n). Figure 3 shows an example of this matrix. Each cell in the matrix contains the conditional probability for each processed term given a particular statement (corresponding to $P(q_k|D_i)$ in the equation). The value of the conditional probability is obtained by applying the dot product operation between all the ϕ values associated with the processed term and all the θ values associated with the statement. The figure depicts an example of this operation, concerning the processed term *text.length* and the statement S_3. The operation is visually represented through a solid orange highlight that points to the result of the operation, stored in the cell of the matrix. The obtained value is the conditional probability for the processed term *text.length* given statement S_3, that is, $P(text.length|S_3) = 0.74$.

Once the matrix is calculated, the conditional probability values of the terms obtained for each statement are used to compute the conditional probability between the query and the different statements as described in the equation. The newly calculated values are the suspiciousness scores associated with each statement. In Figure 3, these values appear in the vector beneath the matrix. The visual representation of this operation is supported by an example, in the form of a light blue highlight of the values obtained for statement S_2, which lead to a conditional probability $P(B|S_2) = 0.35$ between the bug report and statement S_2. This result, $P(B|S_2)$, is stored as the suspiciousness score of S_2.

Finally, the suspiciousness scores are normalized because the values of APR fault localization rankings are between 0 and 1. The outcome is used to obtain the IRFL ranking of suspicious statements, which sorts the statements from the highest to the lowest suspiciousness score. The bottom part of Figure 3 shows an example of the IRFL ranking where S_3 is in the first position of the ranking (hit@1) since it obtains the highest suspiciousness score.

The process that has just been described to obtain the IRFL ranking takes a set of source code files as input. To that end, we set a parameter f representing the number of most suspicious files that should be selected for statement level FL. This file localization, computed on all the files of a given project, follows the exact same steps that are previously presented, but using files as documents instead of statements.

3.2. Combining SBFL and IRFL results

Once the IRFL and SBFL rankings are obtained, the last step is to combine them. We favor two main combination strategies. The first one is a statement-based average, labeled OTM$_{Avg}$. For each of the statements, the combined ranking is obtained by calculating the average of the two suspiciousness scores from the IRFL ranking and SBFL ranking as shown in the upper part of Figure 4.

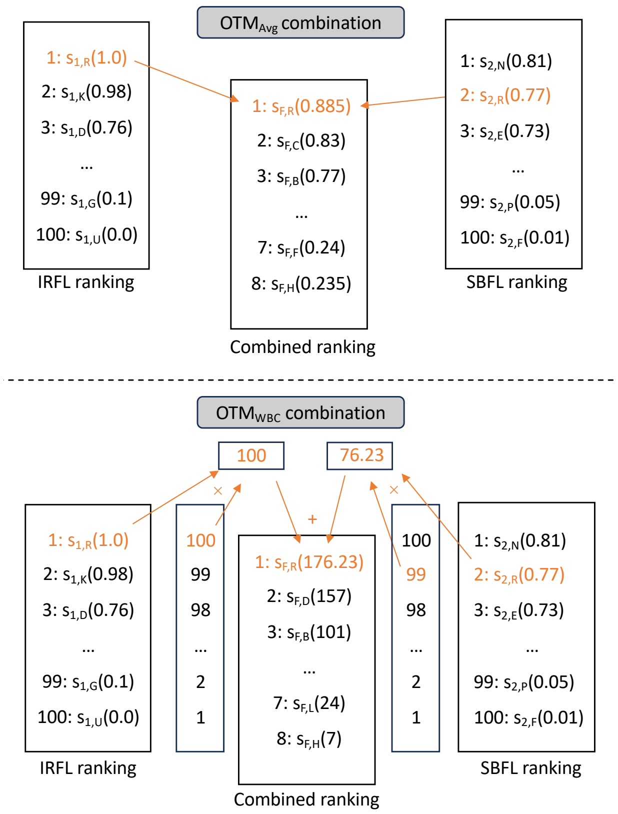

Figure 4: Example of score combination for the IRFL and SBFL rankings using the two variants

The second combination strategy, inspired by the Borda Count family [16], is labeled as OTM$_{WBC}$. This variant takes into consideration the position in the ranking in contrast to the first combination strategy. Thus, it is possible to give weight to the IRFL/SBFL score depending on its position in the ranking. This avoids that similar suspiciousness values are obtained when two statements in the boundaries are combined (e.g., one statement has a suspiciousness value of 1 in the IRFL ranking and a value of 0 in the SBFL ranking), or when statements that have similar suspiciousness scores in the middle of the ranking are combined. The value of each statement is calculated as $rankingSize - stmtRank$ where $rankingSize$ is the size of the two rankings and $stmtRank$ is the rank of the statement in the current ranking (starting at 0). Then, a weight is applied to both values, which is the suspiciousness of their respective statement. Finally, the results are summed pairwise, normalized and sorted to get the final ranking as shown in the lower part of Figure 4. For instance, $s_{1,R}$ and $s_{2,R}$ are the statements for each of the two rankings for location R. Their suspiciousness scores are 1 and 0.77 respectively, their weights are 100 and 99. Indeed, in our example, the rankings have a length of 100 and the two statements are placed first and second (eg. $100-0$, $100-1$). Each value is multiplied by its weight, $1 \times 100 = 100$, and $99 \times 0.77 = 76.23$. The final score is obtained by summing the two products (176.23). The process is repeated for all statements of the ranking.

At this point, it is important to highlight that most of the time, the IRFL and SBFL rankings do not contain the same statements. Some of the statements are present in both rankings, whereas others are present in only one of them. To overcome this issue, any statement contained in only one of the two rankings is added to the other one. Thus, the two rankings have the same size and contain the same statements for the combination. The suspiciousness score of each added statement is set to 0.

Furthermore, it is important to note that the combined ranking can be considered by the developers as an starting point for an iterative refinement process. From there, developers can either manually tweak the statements if multiple and related faults exist in the source code, or modify the inputs (e.g., the bug report) to execute our approach again if the ranking does not contain relevant statements. The iterative refinement process is typically performed in other works [42] that retrieve text using information retrieval techniques since the results depend on the quality of the queries [41, 43].

4. Evaluation

This section explains the evaluation of our work, the research questions we aim to answer, the evaluation process (including metrics, baselines and statistical analysis that we used to answer each research question) as well as the implementation details.

4.1. Research questions

We seek to answer the following three research questions:

RQ$_1$: *What is the performance in terms of solution quality of our approach and the baselines?*

RQ$_2$: *Is the difference in performance between our approach and the baselines significant? If so, by how much?*

RQ$_3$: *Does our approach locate bugs undetected by the baselines (and vice versa)?*

4.2. Planning and execution

Figure 5 shows an overview of the process followed to answer each research question, which is described as follows:

Answering RQ$_1$: To answer this research question (performance in terms of solution quality), inputs from Defects4J [44] are used. Defect4J is a common Java benchmark, and that has been used in previous fault localization and APR works [45, 5, 1, 6].

We choose bugs from five Defects4J projects: Chart, Lang, Cli, Mockito and Math. Specifically, we select single location bugs where the modifications are applied to consecutive lines (or a sole line) within a single file. Some bugs lack associated bug reports, rendering them unsuitable for IRFL computation, so we exclude them. Hence, our experiment counts a total of 59 bugs. For each project, the inputs are: a bug report described in natural language, the source code, and a set of test cases. These inputs are used to assess the performance of our approach considering the two variants presented in Section 3.2.

To put the performance of our approach into perspective, we set five baselines. The first two baselines are SBFL techniques and computed using GZoltar [2]: Ochiai (Baseline$_1$) and DStar [17] (Baseline$_2$). They are two of the most popular ranking strategies used in the literature [1, 45, 24, 15, 4]. Also, Ochiai is known to be the best performing metric for Object Oriented Programming (OOP) languages and is commonly used in FL and APR studies using Java [5, 1, 6].

The following two baselines are IRFL based. Baseline$_3$ is LDA, which is the first step of our approach. Although the application of LDA for locating faulty statements is novel, we set it as a baseline to compare its performance with our approach, which combines the results of IR (LDA) and SBFL (Ochiai). Blues [1] (Baseline$_4$), is built on top BLUiR [37], a file level fault localization using TF.IDF, and leverages it to obtain statement level FL. It is used as the first step of SBIR [1], which is a recent FL tool that combines IR and SBFL using the Cross-entropy Monte Carlo algorithm, a rank aggregation algorithm and the Spearman's footrule. SBIR is Baseline$_5$.

For each variant of our approach or baseline, a ranking of suspicious statements per bug is obtained as Figure 5 shows. To assess the performance in terms of solution quality, we compare the ranking of suspicious statements of each variant of our approach or baseline with an oracle, which is the approved faulty statement (ground truth) of each bug. This results in the calculation of hit@k, which is the number of bugs that are localized when inspecting the top K program statements in a given ranking. Hit@k (also called Top N) is a well-known fault localization metric that is widely used in past FL works [32, 45, 37, 7] since developers often will stop inspecting program elements if they do not get promising results in the top ranked program statements [46].

Specifically, we obtain results for hit@1, hit@3, hit@5 and hit@10. We selected these values because hit@1, hit@5 and hit@10 are commonly used in evaluations [18, 32, 19, 21]. Additionally, we introduced hit@3 to provide insight into statement ranking improvements with higher precision. The results are presented in Table 5.

In addition to hit@k, we utilize one other metric for each ranking of suspicious statements: Mean Reverse Ranking (MRR)[18, 19, 20, 21, 22]. MRR measures the quality of the ranking by capturing how close to the top of the ranking a target (i.e., faulty) statement is retrieved. MRR is computed as the inverse of the rank of the target statement. If the ranking does not contain the target statement, the MRR value is set to 0. The MRR for all ranking is calculated as follow:

$$MRR = \frac{1}{|R|}\sum_{i=1}^{|R|}\frac{1}{rank_i},\ R \text{ a set of rankings}$$

In our case, we compute the MRR for specific hit@k values, so the sizes of the rankings are 1, 3, 5 and 10, respectively. Note that higher MRR values

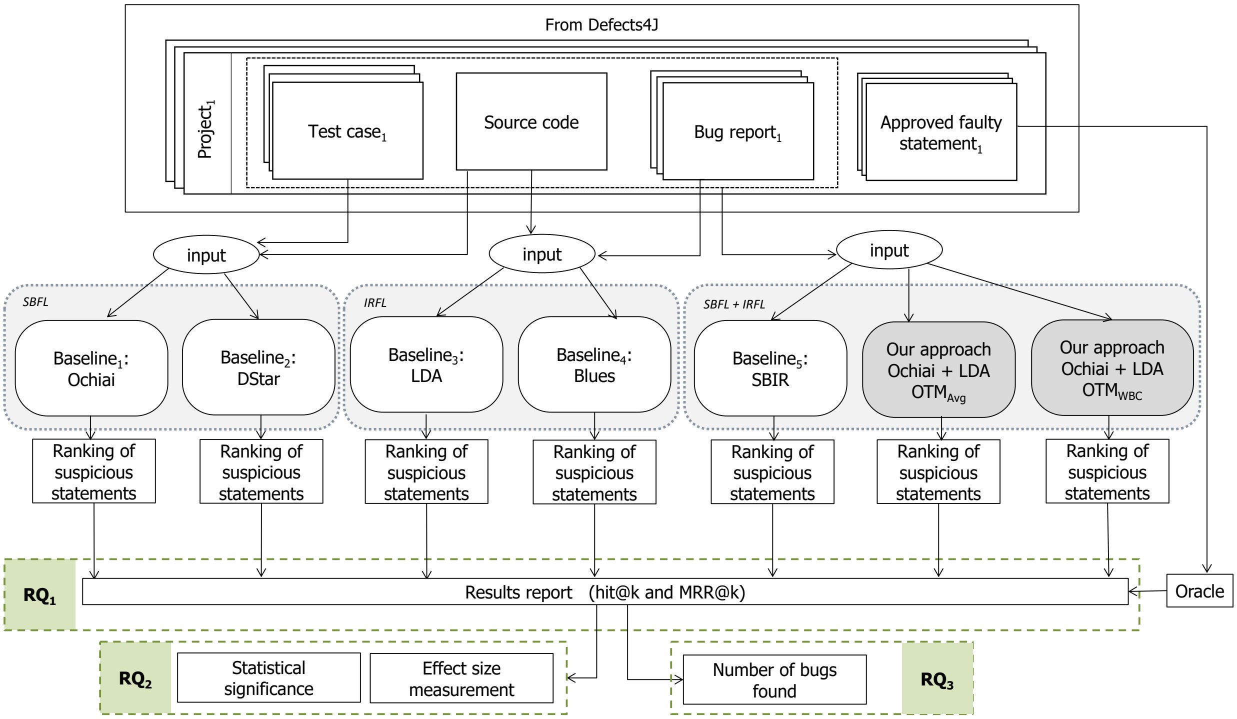

Figure 5: Overview of the evaluation process

indicate better performance.

As suggested by Arcuri and Fraser [47], we executed 30 independent runs per bug when LDA is used (in the two variants of our approach and in Baseline$_3$). Doing multiple executions is commonly done in other FL works from the literature such as [1] that should address random variation. Thus, answering RQ$_1$ entails a total of 1947 runs: 59 bugs x 30 repetitions (Baseline$_3$) + 59 bugs (Baseline$_1$) + 59 bugs (Baseline$_2$) + 59 (Baseline$_4$). The variants and Baseline$_5$ do not require recomputing the values using for the combination as they were already computed by other baselines (Baseline$_1$, Baseline$_3$ and Baseline$_4$). Only the combination of the existing rankings is done each time.

Answering RQ$_2$: To determine whether the difference in performance between of our approach and the baselines is significant, we compare the variant of our approach that obtains the best results in the previous question with each baseline. To achieve this, the results must be properly compared and analyzed using statistical methods. We follow the guidelines outlined in [27] to determine whether differences between the variant of our approach and the baselines are significant. Each comparison yields a corresponding *p-value*. We utilize Holm's post-hoc analysis for pair-wise comparison, ensuring that any differences in results are not merely due to random chance. Our null hypothesis states that the results of our variant are not improved compared to those of the alternative technique. In the research community, a *p-value* below 0.05 implies statistical significance [27]. Thus, a *p-value* below 0.05 allows us to reject our null hypothesis and establish statistical significance.

In case that the difference in performance is significant, it is important to determine through effect size measures how much the results obtained by our approach improve the results obtained by each baseline. For non-parametric effect size measurements, we use Vargha and Delaney's $\hat{A}_{12}$[26]. $\hat{A}_{12}$ measures the probability that running one approach yields higher values than running another approach, so the approaches are compared in pairs (A vs B). If the $\hat{A}_{12}$ statistic obtains a value greater than 0.5, the comparison will be in favor of the A. If the $\hat{A}_{12}$ statistic obtains a value lesser than 0.5, the comparison will be in favor of the B and 1-$\hat{A}_{12}$ will be used to interpret the magnitude of effect. The guidelines for interpreting $\hat{A}_{12}$ values [26] are: the $\hat{A}_{12}$ value of 0.5 means that the two approaches are equivalent (no effect). The $\hat{A}_{12}$ value of 0.56 means a small effect in the magnitude of improvement, 0.64 means a medium effect, and 0.71 means a big effect. For instance, a value of $\hat{A}_{12} = 0.57$ means that on 57% of the runs, approach A would obtain better results than approach B, and that the effect in the magnitude of improvement is small. A value of $\hat{A}_{12} = 0.27$ means that on 73% of the runs, approach B would obtain better results than approach A, and that the effect in the magnitude of improvement is large.

Answering RQ$_3$: To answer this research question, we report the number of bugs identified by our approach's variants and the baselines. Thus, it possible to determine whether our approach locates bugs that the baselines do not and vice versa.

4.3. Implementation details

For the Defects4J benchmark, we use the v2.0.0 version and the Java Development Kit 8. It is important to note that our oracle, which determine faulty statements, is based on the difference between the buggy and fixed source code provided by Defects4J.

As an SBFL tool, we employ GZoltar [2] version 1.7.2 and use both the Ochiai [28] and DStar [17] ranking strategies.

Additionally, we utilize Blues and SBIR to compare our approach to IRFL and combined techniques. These tools were introduced in a previous work [1], which also explores the combination SBFL and IRLF.

In our implementation, we also incorporate JGibblda [48], a Java implementation of LDA using Gibbs Sampling for information retrieval. The values that are set for the LDA parameters that were described in Section 2.3 (k, α and β) may influence the results. For example, if the number of topics (k) is too low, the LDA output may fail to capture the nuanced themes present in the documents. Conversely, if the number of topics is too high, the LDA output may have topics that are too redundant or fragmented. To set the parameter values, Arcuri and Fraser [47] suggest that default parameters are sufficient to measure performance. Hence, we consider the LDA parameter values from a previous work for fault localization using bug reports [39], and after doing experiments, we set the LDA parameters as follows: $k = 400$, $\alpha = 0.01$, and $\beta = 0.01$.

We made a deliberate choice in our preprocessing step. While it is common for IRLF studies to use camelCase splitting as part of their preprocessing [1, 32, 49, 21, 22], we decided to only lowercase the words. After experimenting with various alternatives, we found that this strategy yielded the best result.

In Section 3.1, we introduced the parameter f, which represents the number of most suspicious files considered for statement level localization. Like previous studies [50, 1], f is set to 50.

In order to reproduce the results of this work, the data and source code are available at https://figshare.com/s/99831d5178d72cda36e1.

5. Results

5.1. Research Question 1: Performance

The upper part of Table 1 shows the hit@k values of the baselines and the variants. Both variants outperform all the baselines for all hits. Indeed, in terms of hit@k, both variants obtain the exact same results.

Notably, the results are more favorable for lower hits (i.e. hit@1 and hit@3), indicating that our approach is good at increasing the number of top ranked elements. For instance, both variants show an improvement of 8 bugs for hit@1 compared to Baseline$_1$ and Baseline$_5$, as well as 10 bugs compared to Baseline$_2$ and Baseline$_4$. This represents an improvement of 200% and 500%, respectively. We still have an improvement of 7 bugs compared to LDA

Table 1: Results of the baselines and the variants of our approach

	$k = 1$	$k = 3$	$k = 5$	$k = 10$
	hit@k			
Baseline$_1$ (Ochiai)	4	10	17	22
Baseline$_2$ (DStar)	2	7	10	15
Baseline$_3$ (LDA)	5	6	8	12
Baseline$_4$ (Blues)	2	4	4	7
Baseline$_5$ (SBIR)	4	7	14	20
Our approach-OTM$_{Avg}$	**12**	**19**	**21**	**25**
Our approach-OTM$_{WBC}$	**12**	**19**	**21**	**25**
	MRR@k			
Baseline$_1$ (Ochiai)	0.068	0.107	0.135	0.149
Baseline$_2$ (DStar)	0.034	0.068	0.079	0.090
Baseline$_3$ (LDA)	0.085	0.090	0.097	0.106
Baseline$_4$ (Blues)	0.034	0.048	0.048	0.054
Baseline$_5$ (SBIR)	0.068	0.088	0.113	0.127
Our approach-OTM$_{Avg}$	**0.203**	**0.251**	**0.259**	**0.268**
Our approach-OTM$_{WBC}$	**0.203**	**0.251**	**0.259**	0.267

alone (Baseline$_3$). In fact, 20.34% (12/59) are found at the very top of the ranking using our approach. By only taking into account the bugs that are at hit@10, 48% (12/25) of the bugs are at hit@1. The closer the target statement is to the top of the ranking, the higher the chance of bug detection and resolution.

For hit@3, we observe improvements ranging from 9 (Baseline$_1$) to 15 bugs (Baseline$_4$). 32.20% of the bugs are in the top 3 and 37.29% are in the top 5. Indeed, for hit@5, the improvements ranges from 4 (Baseline$_1$) to 18 bugs (Baseline$_4$) to a total of 21 bugs. Lastly, the improvements for hit@10 range from 3 to 18 with 25 bugs in the top 10, representing 42.37% of all the bugs.

As the lower part of Table 1 shows, the MRR values of our variants also outperform all baselines. For each k, the MRR values of both variants are higher than the MRR values of the baselines. Both variants share the same MRR up to hit@5. For hit@10, OTM$_{Avg}$ is slightly better than OTM$_{WBC}$ even though the difference is minor.

Because the hit@k are equal for the two variants, it is expected for the MRR to be similar. Because MRR takes into account the exact rank for each bug, they provide more precise insights than hit@k. Even though the values are equal for a given hit@k value, the ranks underneath are not always equal. However, they compensate one another and finally obtain the same results.

> **RQ$_1$ answer:** The results reveal that the variants of our approach outperform all baselines in all hit@k and MRR@k values. Specifically, the highest differences between our approach and the baselines are observed in the lower hits (hit@1 and hit@3). This indicates that our approach locates more faulty statements in the lower positions of the ranking.

5.2. Research Question 2: Statistical significance and effect size

Columns 2-3 of Table 2 show the *p-values* of comparing each baseline with OTM$_{Avg}$ (the variant of our approach that obtains the best results). All *p-values* are below the threshold of 0.05. Consequently, we can reject the null hypothesis and establish that our results hold statistical significance.

Table 2: Holm's post hoc *p-values* (statistical significance) and $\hat{A}_{12}$ values (effect size) for each pair-wise comparison

	Statistical significance		Effect size	
	hit@k	MRR@k	*hit@k*	*MRR@k*
Baseline$_1$ (Ochiai) vs OTM$_{Avg}$	0.030	0.030	0.250	0
Baseline$_2$ (DStar) vs OTM$_{Avg}$	0.027	0.030	0.063	0
Baseline$_3$ (LDA) vs OTM$_{Avg}$	0.030	0.030	0.031	0
Baseline$_4$ (Blues) vs OTM$_{Avg}$	0.030	0.030	0	0
Baseline$_5$ (SBIR) vs OTM$_{Avg}$	0.030	0.030	0.188	0

Although Table 2 shows the *p-values* for OTM$_{Avg}$, the *p-values* for OTM$_{WBC}$ are the same since the difference for the MRR at hit@10 for the two variants is minor enough to have no impact on the *p-values.*

Table 2 also shows the $\hat{A}_{12}$ values (effect size) when each baseline is compared with OTM$_{Avg}$. For hit@k (Column 3 of the table), all $\hat{A}_{12}$ values are lesser than 0.5, so the comparison is in favor of our approach. The highest effect size for hit@k is when Baseline$_4$ (Blues) is compared to OTM$_{Avg}$, implying that our approach obtains better results than Baseline$_4$ in 100% of the runs. The smallest $\hat{A}_{12}$ value for hit@k is when Baseline$_1$ (Blues) is compared to OTM$_{Avg}$, implying that our approach obtains better results than Baseline$_4$ in 75% of the runs. For MRR@k (Column 4 of Table 2), all $\hat{A}_{12}$ values are equal to 0, so the comparison is also in favor of our approach. This implies that our approach obtains better results than the baselines in 100% of the runs for MRR@k.

> **RQ$_2$ answer:** We conclude that there are significant differences between our approach and the baselines since all *p-values* are smaller than 0.05. In addition, we conclude that our approach outperforms the baselines by a large effect size.

5.3. Research Question 3: Number of bugs found

Both variants of our approach perform equally with regard to the number of new bugs that are identified. This means that the bugs detected and those missing are the same between the two variants. Consequently, this section refers to the comparison of any of the variants with the baselines.

Using our approach, only 5 bugs are undetected. There are 5 additional bugs that are newly identified by the variants compared to the SBFL techniques, Baseline$_1$ (Ochiai) and Baseline$_2$ (DStar), constituting 8,47% of the bugs. Notably, both Baseline$_1$ (Ochiai) and Baseline$_2$ (DStar) yield identical results regarding the number of missing and detected bugs, as both are computed using GZoltar and thus the same code coverage technique.

Regarding Baseline$_3$ (LDA), our variants identify 16 new bugs. Baseline$_4$ (Blues) detects 7 fewer bugs that the variants, amounting 11,86% of the bugs.

Baseline$_5$ (SBIR) is not compared to the variants in this section because the rankings generated by SBIR only contain the 100 most suspicious statements. Thus, it is impossible to compare the number of missing bugs as it would require having the full rankings.

Since the variants are a combination of Baseline$_1$ (Ochiai) and Baseline$_3$ (LDA), they contain the union of the bugs found in both. Therefore, bugs not identified by Baseline$_1$ (Ochiai) and those missed by Baseline$_3$ (LDA) are all found by the variants. It is important to highlight that the newly found bugs are close to the top of the ranking or even at hit@1 in some instances.

There are 3 bugs that remain undetected by neither the baselines nor the variants. These 3 bugs are consistent across all baselines implying that neither the baselines, being SBFL or IRFL, nor the variants locate these bugs. This accounts for 5% of the bugs. In addition, many of the bugs found undiscovered by either Baseline$_3$ or Baseline$_4$ are common to both.

> **RQ$_3$ answer:** Our approach locates all bugs that are detected by the baselines. In addition, our approach locates 5 bugs that are undetected by Baseline$_1$ (Ochiai) and Baseline$_2$ (DStar), as well as 7 additional bugs in comparison to baseline$_4$ (Blues). In fact, some of these new bugs that are detected by our approach even achieve the top positions of the ranking.

6. Discussion

In this section, we discuss the key takeaways concerning the content of bug reports and their impact on the results.

6.1. Comparison of the results of our approach with the baselines

After analyzing the results, the combination of topic modeling and Ochiai pays off the baselines. The distribution of terms into topics instead of using term frequencies makes that the faulty statement is in a better position of the ranking because it is not necessary that the query contains the exact terms of the statement. If the query contains a term that is in the same topic of a statement, the similarity of that statement is higher than using term frequencies (when the terms query-statement do not match).

In addition, by comparing the results of Baseline$_3$ (LDA) and the variants, we note that combining the SBFL and the IRFL rankings is beneficial. On the one hand, it improves the rank of most of the targets, and thus, more statements are located in the 10 most suspicious statements. When combining two rankings, if the target statement has a high suspicousness in both cases, it will also be the case after the combination, that is not the case for statements that have been considered highly suspicious by a single ranking. Also, if one technique found a statement "particularly" suspicious (i.e. placed at the very top of the ranking), even if the other technique found it only "a little" suspicious, the target will be ranked quite high. On the other hand, by combining two rankings, it allows to browse more different statements. For this reason, more bugs are uncovered by combining rankings in our approach than by using a single technique.

6.2. Takeaways concerning the failing tests and the content of bug reports

Both SBFL and IRFL are limited on the statements they can examine. Because SBFL is based on tests, if there is no failing test passing by the faulty statement, it cannot be detected. Using IRFL, it is important to highlight that the results depend on the quality of the queries as occurs in other works [41, 43]. In our work, the queries are the bug reports, which content can vary significantly depending on the person who authored them. For instance, some reports could be written by the developers of the application, while others would have been written by users that do not know the internal code structure and can only comment on the final behavior of the program. Various factors come into play that can affect the quality of the query, such as the size of the text, the level of precision and detail it provides, and whether or not it includes source code snippets.

Certain bug report patterns are favorable for high performance. For example, if the bug report (i,e., query), contains exactly or almost exactly the target line (document), the similarity between the query and the document is maximal. The statement would thus be ranked close to the top of the ranking.

Nevertheless, bug reports usually contain context (i.e. more than the target line). This may penalize the ranking position of the target statement because other statements highly match with the query due to similar frequencies of terms. This implies that several statements share a nearly identical set of terms or that the query contains enough terms to match other statements with close similarity. Similarly, if the bug report contains source code that is not closely related to the target statement, it would have the same effect. In such cases, some statements would have a high similarity with the query, which would add a lot of noise and move the target statement away from the top of the ranking.

IRFL obtains good results when the bug report contains explanations in natural language but with some programming terms such as class or variable names. The bugs with the best localization performances are not necessarily the ones with the longest bug reports. Long text can contain a lot of context, which would be particularly fitting for file level granularity. However, in the case of statement level FL, the context may increase the similarity of statements that are not the target, even though they may be in the same class or method. Otherwise, although some reports are short, they obtain good results since most of the terms may have relations with the target document.

As IRFL relies on the bug report as the query, bugs without bug reports were excluded from our analysis. However, in some cases bug reports exist but they are nearly empty or consist solely of a title without a description. In such cases, due to the limited terms available for the matching between the query and the documents, the quality of the solution tends to be penalized since the target statement is not in the top positions of the IRFL ranking. For certain bugs where their position in the SBFL ranking is close to the top, the combined ranking helps to improve the quality of the solution.

When poor bug reports obtain extremely bad positions in the IRFL ranking, SBFL results may still struggle to fully compensate the results in the combined ranking. In these situations, the combined ranking does not usually obtain as good results as the SBFL ranking.

With regard to the SBFL ranking, if a SBFL tool does not consider a statement to be faulty, its suspiciousness is assigned a value of zero. However, it is possible that some of these statements may still be faulty despite the tool's verdict (false negative). In

Section 5.3, we examined statements that were absent from the rankings. When considering statements present in the rankings but with suspiciousness scores equal to zero, the results differ. Baseline$_1$ (Ochiai) has two of those statements, and Baseline$_2$ (DStar) has 13. If we categorize them as missing bugs, they represent 15% and almost 34% of all bugs, respectively.

6.3. Undetected bugs

We analyzed why there are three bugs that are not found by any baseline or variant of our approach. In all cases, the required fix is an addition to lines without terms (i.e. a blank line, a closing brace...). In the case of SBFL, there can be no code coverage for a blank line. For IRFL, there is no term to match with the query; thus, blank lines are filtered out of the analysis. This is consistent with the intuition behind the approach and the two variants. In fact, these cases may be almost impossible to solve since there is no prior information helping to deduce that an addition is needed.

7. Threats to validity

We identify four main categories of threats relevant to our work. This classification is based on the suggestion of De Oliveira et al. [51].

Internal validity: In the context of developing our approach, we reuse an existing library [48] to compute LDA in order to avoid implementation issues. Also, we addressed the poor parameter settings threat using values from the literature in our approach and the baselines. Default values are good enough to measure the performance as suggested by Arcuri and Fraser [47]. Nevertheless, we cannot yet claim if the values of the parameters (e.g., the number of topics: k) should be tuned for other fault localization benchmarks.

External validity: Our results derive from an experiment on 59 bugs from five different projects, which come from a well-known benchmark [44] that is also used in previous FL works [5, 1, 6]. For most of the benchmark elements, both source code and bug reports are available.

Also, our technique requires both a test suite (to uncover the bug) and a bug report for the target project. For this reason, some Defect4J projects have been excluded from the experiment. Indeed, this requirement is shared by fault localization techniques. In addition, we suppose that there are enough tests to uncover bugs as well as bug reports written with meaningful identifiers that allow the matching with the source code.

Also, the generalization of the results depend on the quality of the queries as occurs in other IR works [41, 43]. Poor bug reports lead to irrelevant statements in the top positions of the ranking. It is also worth noting that the statements and bug reports must use the same terminology to rank relevant statements with IRFL. To narrow the gap between the statements and the bug reports, different NLP techniques (e.g., lowercasing, tokenizers, and stemming) are applied.

Our implementation requires that the target project is part of the Defects4J benchmark. For this reason, our implementation can be updated to evaluate our approach with Java projects from other benchmarks before assuring its generalization.

Construct validity: Finally, threats to construct validity relate to the hit@k metric usage. Nevertheless, it has been used in previous works [32, 45, 37, 7] and is considered to illustrate the quality of a fault localization approach. In addition, we calculated *p-values* to demonstrate that our approach is statistically significant. All baselines and variants are evaluated in the same way to ensure fairness between comparisons.

Conclusion validity: Finally, our approach exploits randomness. In order to limit the discordance between successive executions, we compute the LDA runs 30 times for each bug as advised in the literature [47].

8. Related work

Previous works have addressed FL using a variety of techniques. We focus on SBFL and IRFL since our work combines them. Table 3 presents previous tools that address FL with SBFL (upper part of the table) and IRFL (middle part of the table). The table also includes previous tools that address FL by combining existing SBFL and IRFL approaches (lower part of the table), and our work (the last row of the table).

Table 3: Previous SBFL and IRFL tools

Work	Tool Name	FL	IRFL Technique	Granularity
Compos et al. [2]	GZoltar	SBFL	-	Statement
Xuan and Monperrus [3]	MULTRIC	SBFL	-	Method
Ribeiro et al.. [52]	Jaguar	SBFL	-	Statement
Silva et al.. [53]	FLACOCO	SBFL	-	Statement
Wen et al. [19]	Locus	IRFL	VSM	Code change
Rahman et al. [54]	Blizzard	IRFL	VSM	File
Moreno et al. [55]	Lobster	IRFL	VSM	File
Ye et al. [56]	-	IRFL	VSM	File
Wong et al. [57]	BRTracer	IRFL	rVSM	File
Zhou et al. [7]	BugLocator	IRFL	rVSM	File
Rahman et al. [58]	MBuM	IRFL	rVSM	Method
Youm et al. [49]	BLIA	IRFL	rVSM	File
Saha et al. [37]	BLUiR	IRFL	TF.IDF	File
Motwani and Brun [1]	Blues	IRFL	TF.IDF	Statement
Wang et al. [21]	AmaLgam	IRFL	TF.IDF	File
Wang et al. [22]	AmaLgam+	IRFL	TF.IDF	File
Li et al. [45]	IRBFL	IRFL	TF.IDF	Method
Zhang et al. [59]	FineLocator	IRFL	TF.IDF	Method
Koyuncu et al. [18]	D&C	IRFL	Gradient Boosting	File
Koyuncu et al. [50]	iFixR	IRFL	Gradient Boosting	Statement
Khatiwada et al. [60]	-	IRFL	Combination	File
Almhana et al. [61]	-	IRFL	Multi objective search	Method
Almhana et al. [62]	-	IRFL	Multi objective search	Method
Poshyvanyk et al. [8]	-	IRFL	LSI	Method
Lukins et al. [14]	-	IRFL	LDA	Method
Nguyen et al. [39]	BugScout	IRFL	LDA	File
Motwani and Brun [1]	SBIR	SBFL & IRFL	TF.IDF	Statement
Le et al. [32]	-	SBFL & IRFL	VSM	Method
Our work	-	SBFL & IRFL	LDA	Statement

8.1. SBFL

SBFL is one of the most researched FL family [23]. It relies on the tests available in a program to detect which parts of the source code is most likely to be faulty. Because it leverages code coverage, it can have low granularity and pinpoint specific statements. Previous SBFL tools are shown at the top of Table 3.

Many APR tools [5, 1, 6] are based on SBFL at the statement level mainly through GZoltar [2]. Xuan and Monperrus [3] lated proposed MULTRIC, a tool combining SBFL formulas in order to improve the performance.

Two additional tools that should be mentioned although they are not as widely used as GZoltar are Jaguar [52] and FLACOCO [53]. Jaguar has the particularity to use data-flow in addition to control-flow. Data-flow is scarcely used because of its high execution cost, however, Jaguar leverages a lightweight approach to solve that limit. FLACOCO has been proposed recently. Like Jaguar, it is based on JaCoCo [34] for code coverage. Jacoco is a Java library that releases version regularly, hence, it can be used on recent Java version. That allows Jaguar and JaCoCo to also be usable on recent Java versions which is not the case of GZoltar. FLACOCO, as well as being available as a command-line interface (CLI) tool, it is also embedded in Continuous Integration (CI) as FLACOCOBOT. When a failing pull request (PR) is detected, it comments SBFL information collected using FLACOCO.

A variety of formulas have been proposed (e.g. Ochiai [28], Jacard [28], DStar [17] and Tarantula [30]). GZoltar and Jaguar both propose a set of several formulas whereas FLACOCO only implemented Ochiai but provide an interface to extend the tool if needed.

8.2. IRFL

IRFL is a family of FL techniques that typically analyzes the textual relationships between a query (e.g., a bug report) and program elements (typically files or methods as the middle part of Table 3 shows).

VSM (Vector Space Model) or rVSM (revised Space Model) are the most used IRFL techniques with 36% of the IRFL tools shown in Table 3. It is used in information retrieval and natural language processing (NLP) to represent text as vectors in high-dimension space. In particular, rVSM addresses certain limitations by, for instance, refining document representation or enhancing the weighting strategy. Tools using these techniques are for example:

Locus [19] utilizes software changes to detect bugs. For instance, commit logs can be mined to identify changes that introduced bugs. The localization is narrowed down to code change level. Blizzard [54] classifies bugs reports into categories depending on their content and then apply query reformulation. Lobster [55], uses stack traces extracted from the bug report. In addition to stack traces, BRTracer [57] segments the bug reports to reduce the impact of noise in long reports. Previous works [63, 64] reported that longer files have a higher chance of being faulty, BugLocator [7] is proposed to address that limitation.

TF.IDF (Term Frequency Inverse Document Frequency) is a weighted scheme used in many studies [37, 1, 21, 22, 45, 59]. VSM tools sometimes use TF.IDF, but due to their specific way of representing documents, we have not labeled them as TF.IDF in Table 3.

BLUiR [37] is based on Indri [65], a search engine based on a language model. It uses the built-in TF.IDF Indri model based on BM25 (Okapi model [66]). One of its peculiarities is that it extracts the terms from the Abstract Syntax Tree (AST) and not directly from the source code. All preprocessing is also done by Indri.

Blues [1], is a recent tool built on top of BLUiR. Like BLUiR, it does not require any training data. After the NLP preprocessing, Blues ranks the project files. This is done with the same parameters as BLUiR. Statements are then extracted from the top suspicious files. These are then fed back to BLUiR to produce a ranking of the statements.

AmaLgam [21] and AmaLgam+ [22] are two tools using more information that simply a bug report. Version history can be used for instance taking into account files modified recently, which have more chance to introduce future bugs [67]. Recognizing that developers often work on a subset of the source code, Wang et al. [21, 22] also considered the history of files previously containing errors to help localize a current bug.

Other studies use different localization techniques. For instance: Koyuncu et al. introduced D&C [18], an approach that employs multi-classification to weight the importance of what they refer to as ”features”. These features correspond to the sources of information derived from bug reports and source code. Their work builds upon previous IRFL studies that have incorporated multiple source of information to enhance fault localization precision [7, 37, 21, 68, 49, 19].

D&C is employed by iFixR [50], a statement level fault localization tool. iFixR has a particularity, as it selectively considers a subset of statements as fault localization information based on their type. Their distinction is motivated by Liu et al. findings [69], which demonstrates that certain types of statements were more error-prone than others. iFixR focuses on 5 statements types: *IfStatements*, *ExpressionStatements*, *FieldDeclarations*, *ReturnStatements* and *VariableDeclarations*.

Khatiwada et al. [60] propose to combine four information retrieval methods: TF.IDF, LSI, Jensen-Shannon Model (JSM) and Pointwise Mutual Information(PMI). Almhana et al. [61, 62] use multi-objective search to improve the performance of IRFL by finding the right balancing between minimizing the number of recommended classes and maximizing the the relevance of the result. Ye et al. [56] using domain knowledge combined with bug reports.

Finally, there are few previous tools using topic modeling as IRFL technique. Poshyvanyk et al. [8] leverage Latent Semantic Analysis (LSI) and Scenario Based Probabilistic for feature localization and applied it to fault localization case study. Even though previous works [14, 39] used LDA, their granularity is different, they focus on methods and files, respectively.

Note that many FL tools based on supervised machine learning are not listed in Table 3 as we considered them outside of the scope of this work, which does not require previous training data.

Among the IRFL techniques shown in Table 3, 55% of them have a file granularity, followed by 32% of tools with method granularity. Only one tool is code changes-based, and only two tools are statement based. Hence, there is a lack of IRFL techniques with statement granularity that can serve to enhance the results of APR tools.

8.3. IRFL and SBFL combination

There have been many studies to improve fault localization using several methods (SBFL, IRFL, MBFL). Even though each of them provides good results, they all have their limitations due to their inputs. Both SBFL and MBFL use program spectra and are thus impacted by the quality of the test suite. Likewise, IRFL uses bug reports as input, the quality of which, has a strong impact on the performance of fault localization.

One way to improve the performance without creating an entirely new method is to combine existing ones. There have been a few studies in this direction. Le et al. [32] propose such an approach to combine SBFL and IRFL. Their approach is split into four components AML^{Spectra}, AML^{Text}, AML^{SuspWord} and Integrator. They respectively focus on calculating the suspiciousness leveraging program spectra, bug reports and words contained in both spectra and the bug reports. The integrator is tasked with combining the result of the three components. The suspiciousness calculated by AML is done at the method level.

Recently, Motwani and Brun [1] proposed SBIR, a tool combining SBFL and IRFL. The rankings come from GZoltar employing the Ochiai strategy and Blues (their own IRFL tool), respectively. The results of the two rankings are combined to produce a final ranking of 100 statements.

The last row of Table 3 compares our work with previous FL works. To the best of our knowledge, this work is the first effort in the literature that investigates fault localization at statement level of granularity using LDA, and combining LDA and Ochiai.

9. Conclusion and future work

This paper proposes a novel approach that combines topic modeling with Ochiai to enhance fault localization at the statement level of granularity. By leveraging the benefits of both Spectrum-Based Fault Localization (SBFL) and Information Retrieval Fault Localization(IRFL), our approach demonstrates significant improvements in fault localization compared to using Ochiai independently. Specifically, using the two variants of our approach, 20.34% of the total number of bugs are ranked in hit@1, compared to only 8.47% for topic modeling (Baseline$_3$), 6.77% for Ochiai and SBIR (Baseline$_1$ and Baseline$_5$) and 3.38% for Blues (Baseline$_4$).

These findings emphasize the potential of integrating topic modeling techniques with existing fault localization approaches, providing valuable insights for the automated program repair community.

As future work, we plan to study the impact on the solution quality using additional sources of information (e.g. stack traces in bug reports). We also plan to study whether extending the preprocessing step (e.g., using automatic query reformulations) can improve the quality of the located statements.

Acknowledgements

This work was supported in part by the Ministry of Economy and Competitiveness (MINECO) through the Spanish National R+D+i Plan and ERDF funds under the Project VARIATIVA under Grant PID2021-128695OB-I00 and in part by the Gobierno de Aragón (Spain) (Research Group S05_20D).

References

[1] M. Motwani, Y. Brun, Better automatic program repair by using bug reports and tests together, in: International Conference on Software Engineering (ICSE), 2023.

[2] J. Campos, A. Riboira, A. Perez, R. Abreu, Gzoltar: an eclipse plug-in for testing and debugging, in: Proceedings of the 27th IEEE/ACM international conference on automated software engineering, 2012, pp. 378–381.

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

[4] K. Liu, A. Koyuncu, T. F. Bissyandé, D. Kim, J. Klein, Y. L. Traon, You cannot fix what you cannot find! an investigation of fault localization bias in benchmarking automated program repair systems, 2019 12th IEEE Conference on Software Testing, Validation and Verification (ICST) (2018) 102–113.

[5] Y. Yuan, W. Banzhaf, Arja: Automated repair of java programs via multi-objective genetic programming, IEEE Transactions on software engineering 46 (2018) 1040–1067.

[6] Z. Chen, S. Kommrusch, M. Tufano, L.-N. Pouchet, D. Poshyvanyk, M. Monperrus, Sequencer: Sequence-to-sequence learning for end-to-end program repair, IEEE Transactions on Software Engineering 47 (2019) 1943–1959.

[7] J. Zhou, H. Zhang, D. Lo, Where should the bugs be fixed? more accurate information retrieval-based bug localization based on bug reports, in: 2012 34th International conference on software engineering (ICSE), IEEE, 2012, pp. 14–24.

[8] D. Poshyvanyk, A. Marcus, V. Rajlich, Y.-G. Gueheneuc, G. Antoniol, Combining probabilistic ranking and latent semantic indexing for feature identification, in: 14th IEEE International Conference on Program Comprehension (ICPC'06), 2006, pp. 137–148. doi:10.1109/ICPC.2006.17.

[9] E. Linstead, P. Rigor, S. Bajracharya, C. Lopes, P. Baldi, Mining concepts from code with probabilistic topic models, in: Proceedings of the 22nd IEEE/ACM International Conference on Automated Software Engineering, ASE '07, Association for Computing Machinery, New York, NY, USA, 2007, p. 461–464. URL: https://doi.org/10.1145/1321631.1321709. doi:10.1145/1321631.1321709.

[10] G. Maskeri, S. Sarkar, K. Heafield, Mining business topics in source code using latent dirichlet allocation, in: Proceedings of the 1st India Software Engineering Conference, ISEC '08, Association for Computing Machinery, New York, NY, USA, 2008, p. 113–120. URL: https://doi.org/10.1145/1342211.1342234. doi:10.1145/1342211.1342234.

[11] F. Pérez, R. Lapeña, A. C. Marcén, C. Cetina, Topic modeling for feature location in software models: Studying both code generation and interpreted models, Information and Software Technology 140 (2021) 106676.

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

[13] H. Jelodar, Y. Wang, C. Yuan, X. Feng, Latent dirichlet allocation (LDA) and topic modeling: models, applications, a survey, Multimedia Tools and Applications 78 (2019) 15169–15211.

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

[15] D. Zou, J. Liang, Y. Xiong, M. D. Ernst, L. Zhang, An empirical study of fault localization families and their combinations, IEEE Transactions on Software Engineering 47 (2021) 332–347.

[16] J. d. Borda, Mémoire sur les élections au scrutin, Histoire de l'Académie Royale des Sciences (1781).

[17] W. E. Wong, V. Debroy, R. Gao, Y. Li, The dstar method for effective software fault localization, IEEE Transactions on Reliability 63 (2013) 290–308.

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

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

[20] X. Ye, R. Bunescu, C. Liu, Mapping bug reports to relevant files: A ranking model, a fine-grained benchmark, and feature evaluation, IEEE Transactions on Software Engineering 42 (2015) 379–402.

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

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

[23] W. E. Wong, R. Gao, Y. Li, R. Abreu, F. Wotawa, A survey on software fault localization, IEEE Transactions on Software Engineering 42 (2016) 707–740.

[24] S. Pearson, J. Campos, R. Just, G. Fraser, R. Abreu, M. D. Ernst, D. Pang, B. Keller, Evaluating & improving fault localization techniques, University of Washington Department of Computer Science and Engineering, Seattle, WA, USA, Tech. Rep. UW-CSE-16-08-03 (2016).

[25] E. Wong, T. Wei, Y. Qi, L. Zhao, A crosstab-based statistical method for effective fault localization, in: 2008 1st international conference on software testing, verification, and validation, IEEE, 2008, pp. 42–51.

[26] A. Vargha, H. D. Delaney, A critique and improvement of the cl common language effect size statistics of mcgraw and wong, Journal of Educational and Behavioral Statistics 25 (2000) 101–132.

[27] A. Arcuri, L. Briand, A hitchhiker's guide to statistical tests for assessing randomized algorithms in software engineering, Softw. Test. Verif. Reliab. 24 (2014) 219–250.

[28] R. Abreu, P. Zoeteweij, A. J. Van Gemund, On the accuracy of spectrum-based fault localization, in: Testing: Academic and industrial conference practice and research techniques-MUTATION (TAICPART-MUTATION 2007), IEEE, 2007, pp. 89–98.

[29] D. Yang, Y. Qi, X. Mao, Evaluating the strategies of statement selection in automated program repair, in: Software Analysis, Testing, and Evolution: 8th International Conference, SATE 2018, Shenzhen, Guangdong, China, November 23–24, 2018, Proceedings 8, Springer, 2018, pp. 33–48.

[30] J. A. Jones, M. J. Harrold, Empirical evaluation of the tarantula automatic fault-localization technique, in: Proceedings of the 20th IEEE/ACM international Conference on Automated software engineering, 2005, pp. 273–282.

[31] F. Steimann, M. Frenkel, R. Abreu, Threats to the validity and value of empirical assessments of the accuracy of coverage-based fault locators, in: Proceedings of the 2013 International Symposium on Software Testing and Analysis, 2013, pp. 314–324.

[32] T.-D. B. Le, R. J. Oentaryo, D. Lo, Information retrieval and spectrum based bug localization: Better together, in: Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering, 2015, pp. 579–590.

[33] T.-D. B. Le, D. Lo, F. Thung, Should i follow this fault localization tool's output? automated prediction of fault localization effectiveness, Empirical Software Engineering 20 (2015) 1237–1274.

[34] M. R. Hoffmann, B. Janiczak, M. E, M. Friedenhagen, Jacoco, 2009. URL: https://www.jacoco.org/jacoco/.

[35] S. Christou, cobertura, 2015. URL: https://cobertura.github.io/cobertura/.

[36] Q. Wang, C. Parnin, A. Orso, Evaluating the usefulness of ir-based fault localization techniques, in: Proceedings of the 2015 international symposium on software testing and analysis, 2015, pp. 1–11.

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

[38] A. Mahmoud, N. Niu, On the role of semantics in automated requirements tracing, Requir. Eng. 20 (2015) 281–300.

[39] A. T. Nguyen, T. T. Nguyen, J. Al-Kofahi, H. V. Nguyen, T. N. Nguyen, A topic-based approach for narrowing the search space of buggy files from a bug report, in: 2011 26th IEEE/ACM International Conference on Automated Software

Engineering (ASE 2011), IEEE, 2011, pp. 263–272.

[40] C. Le Goues, T. Nguyen, S. Forrest, W. Weimer, Genprog: A generic method for automatic software repair, Ieee transactions on software engineering 38 (2011) 54–72.

[41] L. R. Biggers, C. Bocovich, R. Capshaw, B. P. Eddy, L. H. Etzkorn, N. A. Kraft, Configuring latent dirichlet allocation based feature location, Empirical Softw. Engg. 19 (2014) 465–500.

[42] E. Hill, L. Pollock, K. Vijay-Shanker, Automatically capturing source code context of nl-queries for software maintenance and reuse, in: Proceedings of the 31st International Conference on Software Engineering, ICSE '09, 2009, pp. 232–242. doi:10.1109/ICSE.2009.5070524.

[43] B. Sisman, A. C. Kak, Assisting code search with automatic query reformulation for bug localization, in: Proceedings of the 10th Working Conference on Mining Software Repositories, MSR, 2013, pp. 309–318. doi:10.1109/MSR.2013.6624044.

[44] R. Just, D. Jalali, M. D. Ernst, Defects4j: A database of existing faults to enable controlled testing studies for java programs, in: Proceedings of the 2014 international symposium on software testing and analysis, 2014, pp. 437–440.

[45] Z. Li, X. Bai, H. Wang, Y. Liu, Irbfl: an information retrieval based fault localization approach, in: 2020 IEEE 44th Annual Computers, Software, and Applications Conference (COMPSAC), IEEE, 2020, pp. 991–996.

[46] C. Parnin, A. Orso, Are automated debugging techniques actually helping programmers?, in: Proceedings of the 2011 International Symposium on Software Testing and Analysis, ISSTA '11, Association for Computing Machinery, New York, NY, USA, 2011, p. 199–209. URL: https://doi.org/10.1145/2001420.2001445. doi:10.1145/2001420.2001445.

[47] A. Arcuri, G. Fraser, Parameter tuning or default values? an empirical investigation in search-based software engineering, Empirical Software Engineering 18 (2013) 594–623.

[48] P. Xuan-Hieu, N. Cam-tu, Jgibblda, 2008. URL: https://sourceforge.net/projects/jgibblda/.

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

[50] A. Koyuncu, K. Liu, T. F. Bissyandé, D. Kim, M. Monperrus, J. Klein, Y. Le Traon, ifixr: Bug report driven program repair, in: Proceedings of the 2019 27th ACM joint meeting on european software engineering conference and symposium on the foundations of software engineering, 2019, pp. 314–325.

[51] M. de Oliveira Barros, A. C. Dias-Neto, 0006/2011-threats to validity in search-based software engineering empirical studies, RelaTeDIA (2011).

[52] H. L. Ribeiro, R. P. de Araujo, M. L. Chaim, H. A. de Souza, F. Kon, Jaguar: A spectrum-based fault localization tool for real-world software, in: 2018 IEEE 11th International Conference on Software Testing, Verification and Validation (ICST), IEEE, 2018, pp. 404–409.

[53] A. Silva, M. Martinez, B. Danglot, D. Ginelli, M. Monperrus, Flacoco: Fault localization for java based on industry-grade coverage, arXiv preprint arXiv:2111.12513 (2021).

[54] M. M. Rahman, C. K. Roy, Improving ir-based bug localization with context-aware query reformulation, in: Proceedings of the 2018 26th ACM joint meeting on European software engineering conference and symposium on the foundations of software engineering, 2018, pp. 621–632.

[55] L. Moreno, J. J. Treadway, A. Marcus, W. Shen, On the use of stack traces to improve text

retrieval-based bug localization, in: 2014 IEEE International Conference on Software Maintenance and Evolution, IEEE, 2014, pp. 151–160.

[56] X. Ye, R. Bunescu, C. Liu, Learning to rank relevant files for bug reports using domain knowledge, in: Proceedings of the 22nd ACM SIGSOFT international symposium on foundations of software engineering, 2014, pp. 689–699.

[57] C.-P. Wong, Y. Xiong, H. Zhang, D. Hao, L. Zhang, H. Mei, Boosting bug-report-oriented fault localization with segmentation and stack-trace analysis, in: 2014 IEEE international conference on software maintenance and evolution, IEEE, 2014, pp. 181–190.

[58] S. Rahman, K. Sakib, An appropriate method ranking approach for localizing bugs using minimized search space., in: ENASE, 2016, pp. 303–309.

[59] W. Zhang, Z. Li, Q. Wang, J. Li, Finelocator: A novel approach to method-level fine-grained bug localization by query expansion, Information and Software Technology 110 (2019) 121–135.

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

[61] R. Almhana, W. Mkaouer, M. Kessentini, A. Ouni, Recommending relevant classes for bug reports using multi-objective search, in: Proceedings of the 31st IEEE/ACM international conference on automated software engineering, 2016, pp. 286–295.

[62] R. Almhana, M. Kessentini, W. Mkaouer, Method-level bug localization using hybrid multi-objective search, Information and Software Technology 131 (2021) 106474.

[63] N. E. Fenton, N. Ohlsson, Quantitative analysis of faults and failures in a complex software system, IEEE Transactions on Software engineering 26 (2000) 797–814.

[64] T. J. Ostrand, E. J. Weyuker, R. M. Bell, Predicting the location and number of faults in large software systems, IEEE Transactions on Software Engineering 31 (2005) 340–355.

[65] T. Strohman, D. Metzler, H. Turtle, W. B. Croft, Indri: A language model-based search engine for complex queries, in: Proceedings of the international conference on intelligent analysis, volume 2, Washington, DC., 2005, pp. 2–6.

[66] S. E. Robertson, S. Walker, M. Beaulieu, Experimentation as a way of life: Okapi at trec, Information processing & management 36 (2000) 95–108.

[67] S. Kim, T. Zimmermann, E. J. Whitehead Jr, A. Zeller, Predicting faults from cached history, in: 29th International Conference on Software Engineering (ICSE'07), IEEE, 2007, pp. 489–498.

[68] S. M. Wong, W. Ziarko, P. C. Wong, Generalized vector spaces model in information retrieval, in: Proceedings of the 8th annual international ACM SIGIR conference on Research and development in information retrieval, 1985, pp. 18–25.

[69] K. Liu, D. Kim, A. Koyuncu, L. Li, T. F. Bissyandé, Y. Le Traon, A closer look at real-world patches, in: 2018 IEEE International Conference on Software Maintenance and Evolution (ICSME), IEEE, 2018, pp. 275–286.