Analyzing Requirements and Traceability Information to Improve Bug Localization

Michael Rath
Technische Universität Ilmenau
Ilmenau, Germany
michael.rath@tu-ilmenau.de

David Lo
Singapore Management University
Singapore
davidlo@smu.edu.sg

Patrick Mäder
Technische Universität Ilmenau
Ilmenau, Germany
patrick.maeder@tu-ilmenau.de

ABSTRACT

Locating bugs in industry-size software systems is time consuming and challenging. An automated approach for assisting the process of tracing from bug descriptions to relevant source code benefits developers. A large body of previous work aims to address this problem and demonstrates considerable achievements. Most existing approaches focus on the key challenge of improving techniques based on textual similarity to identify relevant files. However, there exists a lexical gap between the natural language used to formulate bug reports and the formal source code and its comments. To bridge this gap, state-of-the-art approaches contain a component for analyzing bug history information to increase retrieval performance. In this paper, we propose a novel approach TraceScore that also utilizes projects' requirements information and explicit dependency trace links to further close the gap in order to relate a new bug report to defective source code files. Our evaluation on more than 13,000 bug reports shows, that TraceScore significantly outperforms two state-of-the-art methods. Further, by integrating TraceScore into an existing bug localization algorithm, we found that TraceScore significantly improves retrieval performance by 49% in terms of mean average precision (MAP).

KEYWORDS

Requirements Traceability, Bug Localization, Software Maintenance, Traceability Recovery, Version History, Machine Learning

ACM Reference Format:
Michael Rath, David Lo, and Patrick Mäder. 2018. Analyzing Requirements and Traceability Information to Improve Bug Localization. In *MSR '18: MSR '18: 15th International Conference on Mining Software Repositories, May 28–29, 2018, Gothenburg, Sweden.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3196398.3196415

1 INTRODUCTION

Encountering unintended or unexpected software system behavior is a common phenomena in the development life-cycle. After defect discovery, a *bug report* is filed and handed to a developer for bug fixing. The bug report provides information about the abnormal behavior and initially guides the developer in retrieving source code files to be modified for removing the defect. Manually scanning the projects code base is time consuming and prone to errors. Extensive project knowledge is required to establish a semantic connection between a reported bug and related source code files. Therefore, it is desirable to automate this process.

Ideally, bug reports should guide a developer in efficiently fixing a program misbehavior. However, the provided information in the report and the one expected and needed by a developer often considerably differ [5, 6]. Furthermore, the developers need to bridge the *lexical gap* [35] between natural language bug description and formal representation found in source code.

State-of-the-art methods analyze multiple development resources to gather relevant information for locating source code files to be fixed for a new bug report. Available algorithms contain dedicated logical components each responsible to handle one of these resources. Typical components are source code structure analysis, version history analysis, and project meta-data analysis. Each component computes a ranked list of source code files ordered by their assumed to the bug report. Finally, these lists are combined to an overall ranked list. On top of the list is the source code file, which most likely needs modification in order to address the bug report.

While the source code structure analysis, the project meta-data analysis, and the results' aggregation received much attention and improvements in previous research, the version history analysis remains less studied. The fundamental idea of project history analysis is using previously resolved bug reports as resource for finding analogies to a current one [26, 58, 61]. The set of source code files modified to resolve previous bug reports is considered candidate to be fixed for a current similar bug report. These previously resolved bug reports are selected based on textual similarity to the new bug report. In this scenario, the lexical mismatch is considerably reduced since the compared texts share the same characteristics, e. g., being informal and written from a user perspective.

We argue that the component responsible for version history analysis could be considerably improved by utilizing two additional information resources. First, instead of just bug reports, the whole history of a development process should be leveraged. This enables proposing not only source code files that already have been part of a bug fix, but rather all source code files modified by project activities become available. Second, in modern software system development, related artifacts are kept in common repositories [45]. Relations between artifacts are captured and maintained as trace links [47, 49]. These trace links allow navigating among different artifacts [25, 29] and among different versions of an artifact. For example, after implementing a requirement or improvement, a trace link to the modified source files is created [46]. The same applies

MSR '18, May 28–29, 2018, Gothenburg, Sweden

ACM ISBN 978-1-4503-5716-6/18/05…$15.00
https://doi.org/10.1145/3196398.3196415

to bug fixes, i. e. all source files for all previously fixed bugs can be traced. Leveraging this information, the second resource in our approach, related artifacts can be detected independently from their textual representation. As complete traceability is difficult to maintain, textual similarity can be used to establish missing links automatically.

In this paper, we study the effectiveness of utilizing additional development artifacts (i. e., requirements and trace links) and their history for bug localization. We propose a version history component called *TraceScore* based on a novel calculation scheme, which uses a bug report, existing project history, and traceability information as input and recovers traceability between this new bug report and existing source code files as output. The result is a ranked source file list with the most relevant files on top.

We conducted a large empirical study on 15 open-source projects containing more than 13,000 bug reports in total in order to answer the following research questions:

RQ-1: How effective is our approach *TraceScore* for bug to source code trace recovery?

RQ-2: How do requirement artifacts and explicit trace links affect *TraceScore*?

RQ-3: What is the impact of filtering historical artifact data on *TraceScore*?

RQ-4: Do bug localization algorithms benefit from *TraceScore*?

Our results are compared to two state-of-the-art version history components: *SimiScore* first proposed by Zhou et al. [61] and also applied unchanged in successive approaches [26, 50, 53, 54, 56] and *CollabScore* by Ye et al. [58], which is also used in [59]. TraceScore significantly outperforms both algorithms in all tested measures, notably by increasing Top-1 by 35.9%, MAP by 37.4% and MRR by 25.1% compared to closest competitor SimiScore. Integrating the TraceScore component into full bug localization algorithm *AmaLgam* [53] doubles Top-1 and significantly increases MAP by 49.6%. The evaluation results confirm that requirements information supports the bug report to source code traceability recovery process.

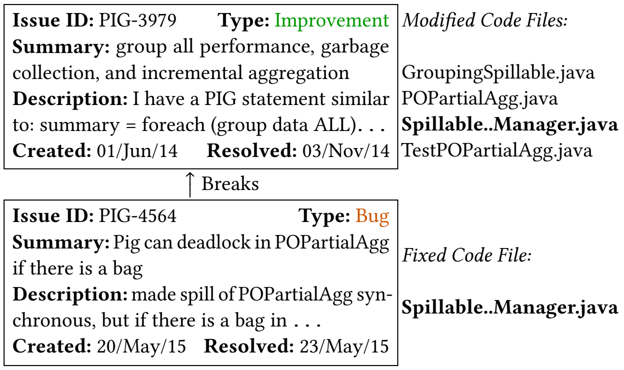
The paper is organized as follows. Section 2 shows a motivating example and introduces the structure of bug localization algorithms focusing on contained version history components. In Section 3, we explain necessary concepts of our approach and describe TraceScore in Section 4. Section 5 studies our approach on 15 open-source projects. The study results are discussed in Section 6. Potential threats to the validity of our study and how we mitigated them are elaborated in Section 7. Related work is presented in Section 8. The paper concludes with Section 9 and outlines further research.

2 VERSION HISTORY COMPONENTS IN BUG LOCALIZATION ALGORITHMS

This section illustrates an example and motivates our approach in the context of related work.

2.1 Motivating Example

The bottom of Figure 1 shows a resolved bug report `PIG-4564`[1] from project Pig. The report consists of a unique issue id, a short textual summary, a longer detailed textual description, and two timestamps: creation and resolution, whereas the latter would be unset for new bugs. Based on the provided information, the task of bug localization is to create a ranked list of source code files relevant for the bug report. Using this list, a developer can investigate the files from the beginning of the list and quickly identify relevant ones instead of searching the complete code base of the project. In the example, one file was modified to fix the bug, which ideally would be on top of the list.



Issue ID: PIG-3979 **Type:** Improvement
Summary: group all performance, garbage collection, and incremental aggregation
Description: I have a PIG statement similar to: summary = foreach (group data ALL). . .
Created: 01/Jun/14 **Resolved:** 03/Nov/14

Modified Code Files:
GroupingSpillable.java
POPartialAgg.java
Spillable..Manager.java
TestPOPartialAgg.java

Issue ID: PIG-4564 **Type:** Bug
Summary: Pig can deadlock in POPartialAgg if there is a bag
Description: made spill of POPartialAgg synchronous, but if there is a bag in . . .
Created: 20/May/15 **Resolved:** 23/May/15

Fixed Code File:
Spillable..Manager.java

Figure 1: Example requirement, bug report, along with modified source code files to implement and fix them.

Figure 1 also shows requirement `PIG-3979`[2] implemented in four files, before the bug was created. The requirement is *explicitly* linked to `PIG-4564` in the projects’ issue tracking system (ITS). This provides valuable information for bug localization, because the implementation of `PIG-3979` modified `SpillableMemory Manager.java`, which was later modified to fix the bug.

Our novel approach TraceScore leverages requirement artifacts and trace links found in ITS to improve bug localization. Since explicit trace links are not always present among issues, artificial ones are established using information retrieval (IR) techniques.

2.2 Components of a Bug Localization Method

State-of-the-art bug localization methods analyze different input resources to create a ranked list of source code files (see Figure 2). Internally, each input resource is processed by a dedicated component and creates an individual ranking. Eventually, these rankings are aggregated by a **composer component** generating the final source code file ranking. Multiple aggregation methods are used to combine the ranking result list of the bug localization algorithms’ components. This includes summation [50], empirically determined weighting schemes [53, 61], trained weights by vector support machines [58] and neural networks [26].

The **source code structure component** relates a given bug report to a project’s code base. It analyzes the source code and typically extracts identifier names and comments. Afterwards, the collected data per source code file and the newly reported bug report are treated as text documents. This allows the application of IR algorithms to compute similarity scores for source code file and bug report combinations [33]. The code structure component creates a source code file ranking using this similarity value as ordering

[1] PIG-4564: https://goo.gl/aUR8nV, fixed files: https://goo.gl/z5Lb8s

[2] PIG-3979: https://goo.gl/hFGZ2K, modified files: https://goo.gl/PFmCAi

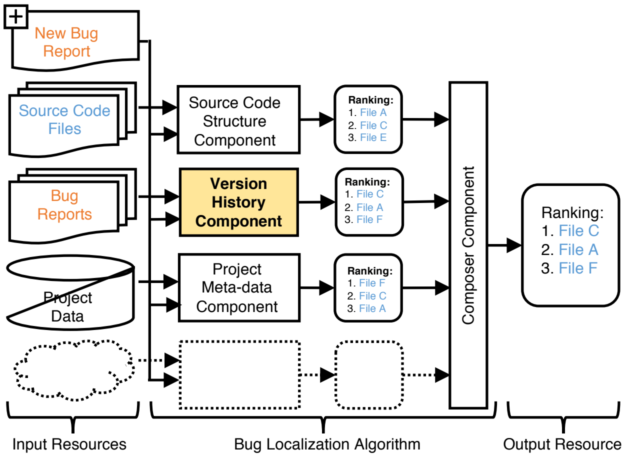

New Bug Report
Source Code Files
Bug Reports
Project Data
Source Code Structure Component
Version History Component
Project Meta-data Component
Ranking: 1. File A 2. File C 3. File E
Ranking: 1. File C 2. File A 3. File F
Ranking: 1. File F 2. File C 3. File A
Composer Component
Ranking: 1. File C 2. File A 3. File F
Input Resources
Bug Localization Algorithm
Output Resource

Figure 2: Structure of a bug localization framework. Our approach, *TraceScore*, provides an evolved algorithm for the version history component, outperforming existing ones.

criteria. Various text matching algorithms have been proposed and studied for this purpose [12].

The **meta-data component** analyzes diverse project meta-data. For example, in [26] the authors utilize developer names and released project version information. Additional, textual resources, e. g., API-documentation, were used to further improve localization results [26, 35, 58, 59].

The **version history component**, the focus of our work, uses previously resolved bug reports and the source code files that were modified in order to fix these. TraceScore implements an evolved algorithm.

2.3 State-of-the-art Algorithms for Version History Component

The version history component of a bug localization algorithm leverages information about previously fixed bug reports, i. e., those reported prior to the new and unresolved bug report. Formally, the component uses the set of all fixed bug reports B and the set of source code files S_B representing all source code files modified in order to fix these bugs $b \in B$. Based on this information and a new and unresolved bug $b^* \notin B$, the component calculates a score for each $s \in S_B$. The score facilitates a ranking with the most relevant source code files on top. It is important to emphasize that $S_B \subseteq S$, meaning that S_B might not represent all source code files in the code base S at the time when b^* is filed and the localization is being performed. Therefore, the search space of the history component is inherently limited to source files, which at least were part of one previous bug fix. However, bugs occur in burst and source files recently fixed are likely to be responsible for new bugs [22]. Without explicitly analyzing the code base, the task the code structure component is responsible for, our proposed approach still increases the search space.

Zhou et al. [61] propose *BugLocator* featuring the version history component *SimiScore*. SimiScore is part of succeeding approaches, namely *BLUiR+* (2013) [50], *BRTracer* (2014) [56], *AmaLgam* (2014) [53], *AmaLgam+* (2016) [54], and *HyLoc* (2015) [26]. Each approach improved the localization performance over previous. Since SimiScore remained unchanged, the increased performance is achieved by advances in other components. SimiScore builds upon the function $fix : b \longrightarrow S_{b,fix} \subseteq S_B$ that returns the set of all source files modified in order to resolve bug report b. If $b^* \notin B$ is a new bug report, the SimiScore per source code file $s \in S_B$ is denoted as:

$$SimiScore(s, b^*, B) = \sum_{\substack{b_i \in \{b_i | b_i \in B \\ \wedge s \in fix(b_i)\}}} \frac{sim(b_i, b^*)}{|fix(b_i)|} \qquad (1)$$

Function *sim* denotes the textual similarity between two bug reports. Thereby, the text of a bug report is formed by concatenating its summary with its description.

Ye et al. [58] proposed *Learning to Rank (LR)*, which utilizes a version history component called *Collaborative Filtering Score (CollabScore)*. CollabScore is also part of *LR + WE* (2016) [59]. The authors define a function $br : s \longrightarrow b_{fix} \subseteq B = \{b_i | s \in fix(b_i)\}$, which calculates the set of all bug reports for which a given source code file $s \in S_B$ was modified. $CollabScore(s, b^*) = sim(b^*, br(s))$ calculates the textual similarity between two bug reports. In contrast to SimiScore, CollabScore only considers bug report summaries and uses a different formula to calculate the textual similarity.

3 BACKGROUND

Our study builds upon the following description model.

3.1 Traceability Information Model

A software engineering process creates manifold development artifacts. Zave et al. [60] propose a "reference model for requirements and specifications" that distinguishes three major artifact types: *requirements specifications*, *design specifications*, and *source code*. Open-source projects rarely use explicit design specification. Therefore we combine the first two artifacts types into one set of requirements denoted with R. S is the set of source code files. Additionally, we introduce a set of *bug reports* B. The definition and implementation of requirements, as well as the discovery of bugs, introduce dependency and implementation links among these artifacts. The function $mod : a \longrightarrow S_a$, with $a \in B \cup R$ and $S_a \subseteq S$ returns the set of modified source files in order to implement/fix artifact a. Figure 3 shows our traceability information model (TIM) and the introduced artifact types. Figure 1 shows typical examples for a bug report and a requirement. In our study, both artifacts are represented equally, only distinguished by the value of *Type* field.

3.2 Trace Path Patterns

All traces originating at bug reports and terminating at code files are relevant for our approach. Figure 4 shows relevant trace path patterns. We consider two path types, each starting at a bug report denoting the new bug to be localized. The first trace link path, $b \xrightarrow{dep} b \xleftarrow{impl} s$ with $b \in B, s \in S$ represents the relation of a bug report via a previously filed bug report (dependency link) to

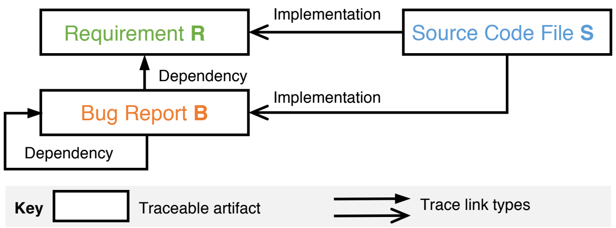


Figure 3: Traceability information model (TIM)

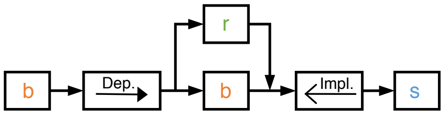


Figure 4: Considered trace path patterns between bug reports *b*, requirements *r* and source code files *s*.

a source code file that was modified to fix the bug. The second trace link path, $b \xrightarrow{dep} r \xleftarrow{impl} s$ with $b \in B, s \in S$ represents the relation from the bug report via a requirement to a source code file implementing that requirement. Using these patterns, all traces from a given bug to all related source code files can be retrieved.

4 THE TRACESCORE APPROACH

TraceScore uses historical project data enriched with previously unused information in order to increase localization performance. Given a bug report b^*, TraceScore creates a ranked list based on relevance of source code files, that are potential candidates to be modified in order to fix b^*. The proposed localization process is shown in Figure 5 and explained in the following paragraphs.

4.1 Step ❶: Selecting Artifacts from Project History

The first step is artifact selection. In contrast to SimiScore and CollabScore, we consider previously resolved bug reports *and* realized requirements, because each modification may introduce new faults into the software. Further, we conditionally filter both the selected artifact types in two ways. The first condition restricts the number of files modified to resolve a bug, $|mod(b)|$, or implement a requirement, $|mod(r)|$, for $b \in B, r \in R$. The rationale is that the larger this number, the smaller is the information gained by an individual traced source code file. The second condition applies filtering on the time domain, which has been reported to be effective before [27]. We exclude artifacts that were resolved more than a defined number of days before b^* was filed. The rationale is that the longer a source code file was not changed, either as part of a bug fix or by implementing a requirement, the more mature its code is. The function $diff(a, b^*)$ calculates the number of days between artifact $a \in B \cup R$ was fixed before b^*. Despite previous studies elaborating a smoothing effect varying over time, we simply use a hard cut off. Eventually, the first step selects two artifact sets

$$R_{sel} = \left\{r \middle| |mod(r)| \leq \mathcal{M}_R \wedge diff(r, b^*) \leq \mathcal{D}_R\right\} r \in R$$
$$B_{sel} = \left\{b \middle| |mod(b)| \leq \mathcal{M}_B \wedge diff(r, b^*) \leq \mathcal{D}_B\right\} b \in B$$

with $\mathcal{M}_R, \mathcal{M}_B$ as upper limits for number of source files per artifact and $\mathcal{D}_R, \mathcal{D}_B$ as upper limits for time difference in days.

4.2 Step ❷: Preprocessing Artifact Texts

We apply commonly used preprocessing steps to the text combined from summary and description to each artifact $a \in R_{sel} \cup B_{sel} \cup \{b^*\}$: stop word removal, camel case splitting, lower casing and stemming [41]. The resulting *bag-of-words* [33] representation per document is used to build a *document-term-matrix* of the artifacts $a \in R_{sel} \cup B_{sel}$. Analogous to *SimiScore* [61], we use the logarithmic $tf \times idf$ term weighting scheme and denote $V(a)$ as vector space representation of artifact a.

4.3 Step ❸: Analyzing Textual Similarity

The vector space representation $V(b^*)$ is considered as query and used to search for relevant artifacts within the corpus. Cosine similarity [33] is used to calculate the similarity of two documents. The result is a similarity score of b^* to every $a \in R_{sel} \cup B_{sel}$.

4.4 Step ❹: Creating a Graph for Analyzing Traceability

In this step, a traceability graph G is created as follows. Its nodes represent the artifacts $\{b^*\} \cup R_{sel} \cup B_{sel} \cup S_{mod}$ with $S_{mod} = \left\{s \middle| s \in \{mod(a) | a \in R_{sel} \cup B_{sel}\}\right\}$. Implementation edges are added between nodes representing source code files and nodes representing requirements and bug reports, according to function *mod*. From b^* a weighted dependency edge is created to every node representing a requirement $r \in R_{sel}$ and a bug report $b \in B_{sel}$. The edge weight is determined by function $weight(a, b^*)$, which is 1.0, in case an explicit dependency trace link exists between a and b^* in the project history, and textual similarity $sim(a, b^*)$ otherwise. During the process of creating a new bug report b^*, the author is able to manually specify other bug reports or requirements this bug relates to. If no such information exists, the textual similarity calculated in step ❸ is used. The construction of the graph assures the existence of a trace path from every node to the node representing b^*.

Figure 6 shows an example of a traceability graph. The depicted graph consists of b^*, the bug report for which the to be modified source code files need to be found, $r_{0...2}$ the selected previously implemented requirements and $b_{0...2}$ the selected previously fixed bug reports. $s_{0...11}$ are source code files modified to implement the requirements and to fix the bugs, i. e. the elements of S_{mod}. At this stage, each source file has the same relevance for b^*. In the next step, a score value per source code file is calculated and then used to rank the files accordingly.

4.5 Step ❺: Calculating TraceScore per Source Code File

TraceScore is an evolved version of SimiScore (see Eq. (1)). To motivate the proposed changes, we first demonstrate why it is not advantageous to simply apply SimiScore to requirement artifacts.

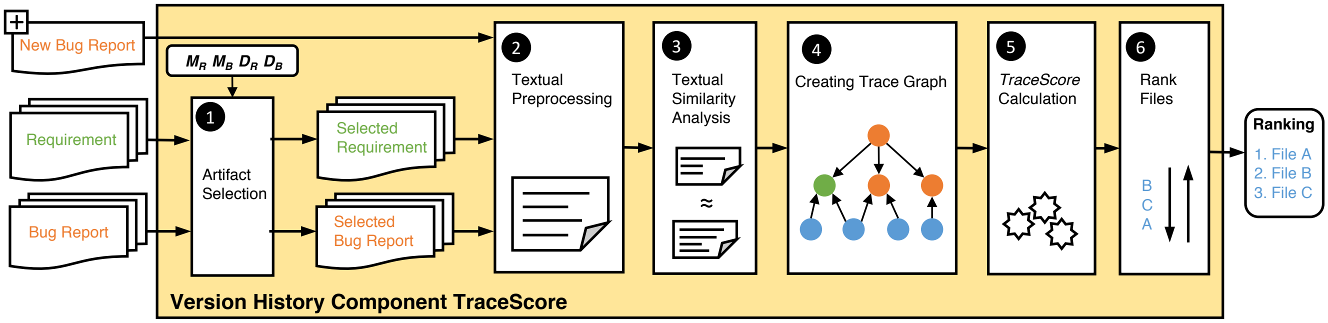


Figure 5: Architecture of our version history component TraceScore. The project's requirement artifacts and the bug report to process are used as input. The individual processing steps are depicted as boxes and arrows denote the flow between the steps. The result is the established traceability from the bug report to the relevant source files.

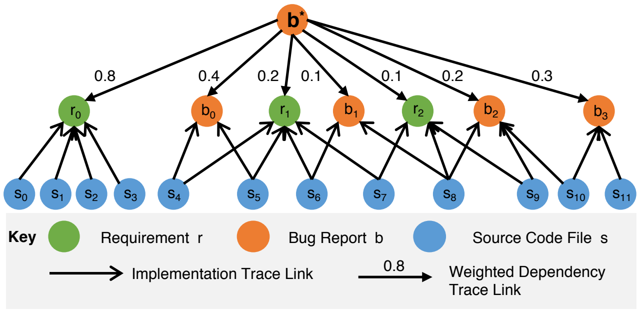


Figure 6: Example traceability graph according to TIM (see Figure 3) created to recover trace link for bug report b^* to source code files.

While the incorporation of requirements artifacts likely extends the localization search space to other potentially defective files, it also increases the chance for retrieving more false positives (see Figure 1). For each source code file s, SimiScore sums up ratios. Each ratio is determined as the textual similarity of a bug report that led to a modification of s divided by the total number of files modified to resolve this bug report. For example, the SimiScore for s_{10} in Figure 6 is $SimiScore(s_{10}, b^*) = \frac{sim(b_2,b^*)}{3} + \frac{sim(b_3,b^*)}{2} = \frac{0.2}{3} + \frac{0.3}{2} \approx 0.22$. Similarly, applying SimiScore to a source code file realizing a requirement, e. g., s_0 results in $SimiScore(s_0, b^*) = \frac{sim(r_0,b^*)}{4} = \frac{0.8}{4} = 0.2$. While s_0 is realizing requirement r_0, which has a very high textual similarity to the current bug report, the source code file s_{10} would still get a higher score due to the fact that it was modified to resolve multiple bug reports though they have far less textual similarity. Additionally, the summation is a linear combination of nonlinear textual similarity values (underlying cosine function is nonlinear), which vanishes the discriminating of the terms. Bug report b^* has a very high textual similarity to requirement r_0, indicating that source code files $s_{0...3}$ need to be modified in order to resolve b^*. Nevertheless, SimiScore would calculate a higher rank for s_{10}, overriding the exceptional high similarity value.

To overcome the reported deficiencies, we derived TraceScore from SimiScore with the following assumptions.

(1) A small number of source code files S_B need to be changed to fix a bug report.
(2) On average, the number of source code files S_R modified to implement a requirement is larger than that for resolving bug reports: $S\_R > S\_B$.
(3) A source code file is more often changed in order to resolve a bug report B_S than for implementing a requirement R_S, i. e., $B\_S > R\_S$.

Based on these assumptions, we define TraceScore for each $s \in S_{mod}$ and $a, a_i \in B_{sel} \cup R_{sel}$ as

$$TraceScore(s, b^*) = \sum_{a_i \in \{a | s \in mod(a)\}} \frac{weight(a_i, b^*)^2}{|mod(a_i)|} \qquad (2)$$

The values S_R and S_B affect the denominators of the formula. Consider two source files, one modified by a requirement r_x and one modified by a bug report b_x, both connected to b^* with same edge weights, i. e: $weight(r_x, b^*) = weight(b_x, b^*) = weight$: Assumption $S\_R > S\_B$ implies, the added ratio for each source code file is larger, if it is connected to a bug report than if it is connected to a requirement: $\frac{weight^2}{S\_B} > \frac{weight^2}{S\_R}$.

The values B_S and R_S affect the number of terms in the formula for each source code file. Combining the previous statements about ratios, and our assumption $B\_S > R\_S$, would result in similar problems of vanishing textual similarity as outlined for SimiScore. We compensate for this by pruning the trace graph and squaring the weights. The square reintroduces nonlinearity, i. e., dampens low textual similarity, while explicit dependency edge weights remain unchanged: $1^2 = 1$. The pruning adds an upper bound to the summation, preventing the accumulation of small ratios to high values. The pruning already occurred in Step ❶. The time parameters $\mathcal{D}_{\mathcal{R}}$ and $\mathcal{D}_{\mathcal{M}}$ limit B_S and R_S. Upper bounds for S_B and S_R are provided by $\mathcal{M}_{\mathcal{B}}$ and $\mathcal{M}_{\mathcal{R}}$.

For the example graph in Figure 6, the TraceScore value for s_0 is $TraceScore(s_0, b^*) = \frac{0.8^2}{4} = 0.16$ and for s_{10} it is $TraceScore(s_{10}, b^*) = \frac{0.2^2}{3} + \frac{0.3^2}{2} \approx 0.058$. Thus our algorithm suggests it is more likely to modify s_0 than s_{10} in order to resolve b^*.

Comparing Equations (1) and (2), the changes seem minor, but show to have a significant influence, as discussed in Section 6.

4.6 Step ❻: Recovering Bug to Source Code Traceability

The last step sorts the source code files $s \in S_{mod}$ in descending order of their respective $TraceScore$ values. This creates a ranked list for bug b^* with the most relevant source code files on top.

5 STUDY DESIGN

In order to answer our four research questions (see Section 1), we conducted a large study with 15 open-source software projects.

5.1 Project Selection

We selected open-source projects based on two criteria. First, since our approach analyzes bug reports, requirements information, and trace links among these artifacts (see Section 4), we only selected projects providing these data. Second, in order to provide a meaningful project history, we only selected projects that are in development for at least five years. Applying these criteria, we selected 15 popular projects mainly from Apache Project and JBoss.

5.2 Data Demographics of Studied Cases

Table 1 shows the characteristics of chosen projects: studied time period, the total number of bug reports and requirements. Additionally, there is a detailed column about the minimum, mean, average, and maximum number of source code files modified in order to fix a bug report and to implement a requirement. The last columns shows the number of dependency trace links explicitly defined by the developers of the respective projects. We retrieved raw data by collecting relevant project artifacts from each project's website. The collection was done on April 7th 2017.

5.3 Data Collection

A two step data collection process was sequentially applied to the 15 selected projects.

Step 1: Analyzing project management and issue tracker system. We implemented a collector to retrieve artifacts (i. e. requirements and bugs) and their trace links. All examined projects use the JIRA [20] project management tool offering a web-service interface. Our collector downloaded and parsed all artifacts and dependency trace links.

Step 2: Analyzing Source Control Management (SCM) system. A second collector was implemented to download all source code changes and commit messages from each SCM repository. All 15 studied projects used Git[15] as SCM. We parsed the commit messages and applied the heuristic described in [3] to discover implementation trace links to bug reports and requirements. Certain modifications of the code base require to alter non-source code files, e. g., documentation, or files for build automation, which are not relevant for bug localization. We excluded these files based on the file name extension. The results of these two steps were stored in a database per project, which is publicly available [44].

5.4 Evaluation Setup

5.4.1 Comparsion with version history components. We compare TraceScore against the two state-of-the-art version history components *SimiScore* [61] and *CollabScore* [58]. Furthermore, we evaluate different parameter configurations for our TraceScore algorithm to study the influence of each parameter in depth (see Table 2). Based on our assumptions described in section 4.5, we empirically determined a baseline configuration *TraceScore (Baseline)* by using the artifact selection criteria $\mathcal{M}_B = 10$, $\mathcal{M}_R = 20$, and $\mathcal{D}_B = \mathcal{D}_R = 365\,days$. Thus, we exclude bug reports that required changing more than 10 source code files as well as requirements that are implemented in more than 20 source code files. Additionally, we excluded artifacts resolved more than one year before the new bug report was issued. In the second configuration, all requirement information is excluded, i. e., $R_{sel} = \emptyset$. The 3rd configuration changes the weight function to $weight(a, b^*) = sim(a, b^*)$, and thus excludes all explicitly defined dependency trace links. The 4th configuration shortens the history to half a year. Finally, configuration 5 does not filter the artifacts based on the number of source code files they affect.

We processed the 15 projects separately, by first ordering the contained bugs by resolution date starting with the oldest one. This list is processed front to back, using the current bug as the one to locate b^*. For this, the SimiScore, CollabScore, and TraceScore in all five configurations are applied and the resulting rankings are captured. Results are evaluated by comparing computed rankings with the list of source code files that were actually modified in order to resolve a bug report.

5.4.2 Automated bug localization algorithm using TraceScore: ABLoTS. We replaced the version history component of AmaLgam [53] with TraceScore, to study its application in a complete bug localization algorithm. AmaLgam is the most advanced algorithm with publicly available source code [2]. It consists of three components, each calculating a *suspiciousness score Susp* for a given bug $b* \in B$ and $s \in S$. $Susp^R(s, b*)$ represents $SimScore(s, b*)$, $Susp^S$ is the structure component taken from BLUiR [50], and $Susp^H(s)$ is taken from BugCache [22]. BugCache predicts future bugs by maintaining a relatively short list of most fault-prone program entities. In AmaLgam, a composer applies two empirically determined weighting factors a and b to the three individual suspiciousness scores to create $Susp^{S,R,H}(s, b*)$ for a code file s used for ranking.

We built ABLoTS using the components of AmaLgam, but replaced SimiScore, i. e. $Susp^R$, with TraceScore in BaseLine configuration. Further, instead of a fixed weighting scheme for the three individual code file scores, we applied a supervised learning classifier for categorizing source file and bug report pairs. We utilized Weka's [17] J48 decision tree with default pruning settings because of its previously reported effectiveness in other software engineering studies [16]. We formed 4-tuples consisting of $Susp^S(s, b)$, $Susp^H(s, b)$, $TraceScore(s, b)$, and a class label C_{true} or C_{false} encoding if $s \in mod(b)$, i. e. if s was modified to resolve b. These instances were used to train and test the classifier. On a project basis, we ordered all bug reports by resolved date and used the first 80% for training and the remaining 20% for testing. Because of bug history, a commonly used 10-fold-cross validation is not applicable. Few source code files are modified to resolve a bug (see Table 1), and thus created training instances were severely unbalanced containing many more instances with C_{false}. Training against such unbalanced sets makes it likely that the classifier will favor placing instances into the majority class, i. e. the file s does not fix b. We

Table 1: Characteristics of the studied cases.

Project	Studied Time Period	#Bug Reports	Changed Source Code Files per Bug Report				#Requirements	Changed Source Code Files per Requirement				#Dependency Trace Links
			min	median	mean	max		min	median	mean	max	
Axis2	2005-07 – 2017-03	1078	1	2	4.4	155	353	1	5	37.4	1144	96
Derby	2004-09 – 2016-12	1778	1	2	6.5	1920	1264	1	4	14.4	1334	1764
Drools	2005-12 – 2017-03	1281	1	3	17.4	1371	653	1	11	76.7	3247	153
Hadoop	2006-06 – 2016-11	748	1	2	3.0	85	746	1	3	15.0	3676	1131
HornetQ	2006-05 – 2015-06	270	1	4	7.4	50	187	1	10	59.5	3680	42
Infinispan	2009-03 – 2016-12	1996	1	3	5.9	167	1468	1	6	24.0	2851	745
Izpack	2009-01 – 2016-01	318	1	3	9.0	140	160	1	8	24.0	460	49
Keycloak	2013-07 – 2017-04	786	1	4	11.3	645	637	1	10	39.8	4652	360
Log4J2	2008-12 – 2017-04	441	1	2	7.6	385	335	1	4	31.7	1266	200
Pig	2008-02 – 2017-04	1265	1	2	4.0	130	623	1	4	9.8	391	513
Railo	2008-11 – 2013-12	300	1	2	4.0	66	129	1	4	8.5	140	7
Seam2	2005-08 – 2014-03	776	1	1	2.7	63	526	1	3	12.8	2268	246
Teiid	2004-04 – 2017-04	1297	1	3	14.0	1073	1162	1	8	72.9	3616	311
Weld	2009-01 – 2017-03	560	1	4	8.7	570	419	1	7	25.0	2050	228
Wildfly	2010-07 – 2016-12	687	1	2	7.7	295	557	1	8	37.9	3270	1925

Table 2: Studied parameter settings of TraceScore (TS).

Id	Configuration Name	$\mathcal{M}_B$	$\mathcal{M}_R$	$\mathcal{D}_B$ [*days*]	$\mathcal{D}_R$ [*days*]
1	TS Baseline	10	20	365	365
2	TS No Requirements	10	0	365	0
3	TS No Explicit Dependencies	10	20	365	365
4	TS Short History	10	20	180	180
5	TS All Source Code Files	∞	∞	365	365

Table 3: Average accuracy for 15 projects of retrieved bug to code traceability by version history components.

Algorithm	$\overline{\text{Top-1}}$	$\overline{\text{Top-5}}$	$\overline{\text{Top-10}}$	$\overline{\text{MAP}}$	$\overline{\text{MRR}}$
SimiScore	0.130	0.283	0.369	0.146	0.208
CollabScore	0.040	0.105	0.150	0.056	0.078
TraceScore	**0.174**	**0.350**	**0.436**	**0.202**	**0.260**

used Weka's inbuilt sub-sampling feature to create balanced data sets. Given a fixed number of instances labeled C_{true}, Weka randomly selects the same number of instances C_{false}. We trained one classifier in turn using the balanced sets for the project and then evaluated the classifier against the respective unbalanced testing set. To mitigate the random effects of sub-sampling, we repeated the training and testing with 10 different J48 classifiers for every project and averaged the achieved results.

5.5 Evaluation Metrics

We use accepted metrics [53, 56, 61] to evaluate the achieved effectiveness of the compared methods. *Top@k* [33] measures the percentage of bug reports for which at least one correct predicted source file is among the top k ranked files. The *mean average precision (MAP)* [33] provides a single measure of quality across multiple query results. For one query, the average precision is the average of the precision value obtained for the set of top K documents in the ranked list. This value is averaged over all queries. The average position of the first relevant document in the ranked list is defined as *mean reciprocal rank (MRR)* in [52].

6 STUDY RESULTS

In this section we answer our research questions one by one.

6.1 How effective is TraceScore for bug to source code trace recovery?

To investigate RQ-1, we measured the effectiveness of "TraceScore Baseline" configuration for 15 open-source projects and compared the performance metrics with SimiScore and CollabScore. TraceScore Baseline outperforms these approaches in almost all metrics. Therefore we only report the averaged metrics across all projects (Table 3) and discuss exceptional cases. Detailed values are part of reproduction data [44]. SimiScore achieves a slightly higher Top-1 value in project Railo and Weld. For project Railo, SimiScore shows competitive results compared to the "TraceScore Baseline" configuration. CollabScore performs worst in all metrics and on every project, likely stemming from its rather simple algorithm only relying on textual similarity of the summary of bug reports (see Section 2.3). On average "TraceScore BaseLine" configuration achieves a 35.9% better Top-1, 23.8% better Top-5, 19% better Top-10, 37.4% better MAP, and 25.1% better MRR value than its closest competitor SimiScore across the 15 projects. An application of non-parametric Kruskal-Wallis test [24] confirms significant ($p < 0.05$) differences among the three algorithms in all studied metrics. Applying Dunns [13] post hoc test with Bonferroni correction shows that these differences stem from TraceScore.

> Finding 1 (RQ-1) TraceScore is effective for recovering traceability information between bug reports and source code files. It outperforms existing version history components in terms of Top-1, Top-5, Top-10, MAP, and MRR.

For bug report Derby-4214[3] of project Derby, SimiScore ranks fixed file `DD_Version.java` as 2$^{\text{nd}}$, whereas TraceScore ranks it

[3] DERBY-4214: https://goo.gl/bdQqCE, fixed files: https://goo.gl/rHVb8p

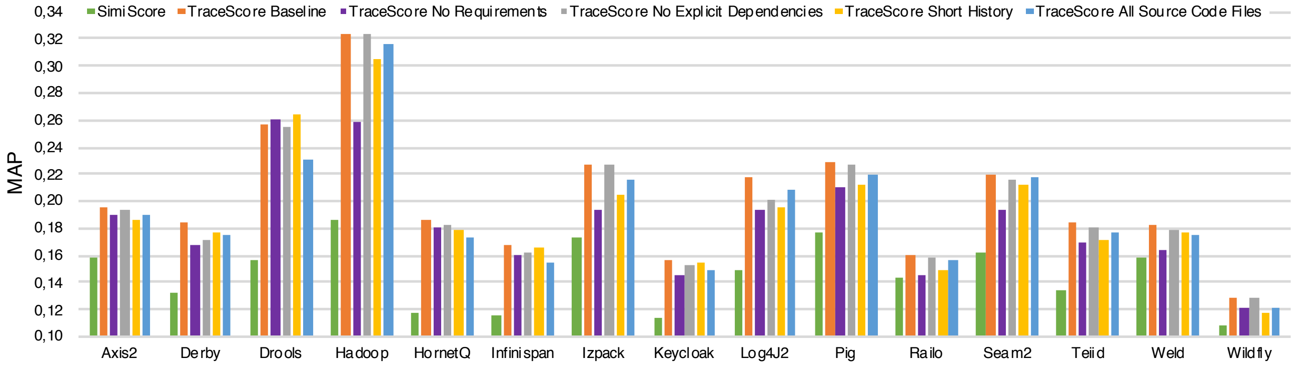


Figure 7: Comparison of SimiScore and different configurations of TraceScore in terms of MAP.

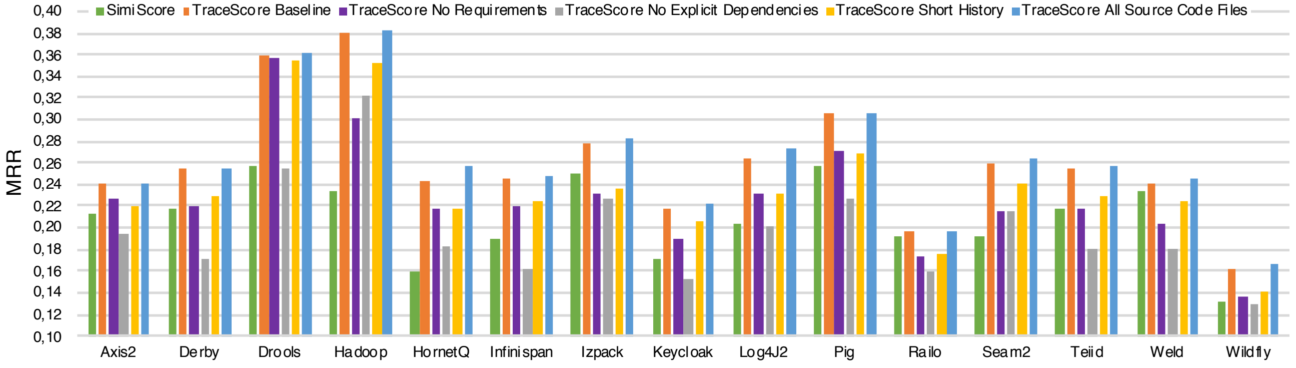


Figure 8: Comparison of SimiScore and different configurations of TraceScore in terms of MRR.

only 4th. However, TraceScore also correctly ranks `DataDictionaryImpl.java` as 1st which was modified prior to implement improvement Derby-3769[4]. Contrasting SimiScore, TraceScore utilizes this information, as well as the existing trace link among the two artifacts and thus achieves a better result.

6.2 How do requirements artifacts and explicit trace links affect TraceScore?

To investigate the influence of requirements information and explicit trace links, we created two configurations of TraceScore derived from “TraceScore Baseline” (see Table 2). The first, “TraceScore No Requirements”, ignores all requirement artifacts, i. e., $R_{sel} = \emptyset$ independent from b^*. The resulting MAP and MRR metrics for all projects are shown in Figure 7, and Figure 8.

For all projects, both the MAP and MRR metric decrease compared to “TraceScore BaseLine”. The highest drop occurs in project Hadoop, where the MAP is reduced by 30% and the MRR by 22%. Ignoring the requirements artifacts for bug localization has nearly no effect on Drools. Indeed, the MAP value even slightly increases. Nevertheless, even without utilizing requirements artifacts, TraceScore achieves a higher accuracy in terms of MAP and MRR as SimiScore, in 14 out of 15 projects. In project Railo, which contains the least number of requirements and explicit dependency links (see Table 1), SimiScore performs better.

The configuration “TraceScore No Explicit Dependencies” uses the same parameters as “TraceScore Baseline”, but ignores explicitly defined dependency trace links extracted during data collection (see Section 5.3). This results in a slight decrease of MAP and a much larger in MRR compared to “TraceScore Baseline”. The interpretation is that the created source code file rankings are shifted downwards, i. e., relevant source code files appear on a lower rank, but the relative order stays the same. This can clearly be seen in project Drools, where the MRR decreased to 25% (i. e, on average the first relevant source code file is on 4th position in the ranked list) compared to 35% in “TraceScore Baseline” (i. e., on average the first relevant source code file is on 3rd position). The MAP metric as well as Top@k are not much affected by such a change in rankings. SimiScore, which does not elaborate explicitly created dependency trace links, is superior to “TraceScore No Explicit Dependencies” in terms of MRR.

Finding 2 (RQ-2) Requirements artifacts and explicitly defined dependency trace links improve the bug localization performance of TraceScore.

6.3 What is the impact of filtering historical artifact data on TraceScore?

In configuration “TraceScore Short History” we set $\mathcal{D}_B$ and $\mathcal{D}_R$ to 180 days: roughly bisecting the values compared to “TraceScore

[4] DERBY-3769: https://goo.gl/vyoFmb, modified files: https://goo.gl/iDJhK5

Baseline" configuration. This negatively effects MAP and MRR (see Figures 7 and 8) on all projects, except Drools. TraceScore loses information by limiting the time interval of previously resolved bug reports and implemented requirements. For example, bug report Derby-6705[5] was created and resolved in August 2014. It has a dependency trace link to bug report Derby-6357[6], which has been fixed in October 2013. This information is no longer available during localization of Derby-6705 if the history is too short.

The last configuration "TraceScore All Source Code Files" studies the influence of not restricting the number of changed files per requirement and bug report, i. e., $\mathcal{M}_B = \mathcal{M}_R = \infty$. In terms of MRR, this change in configuration has no effect compared to "TraceScore Baseline". Thus on average, the rank of the first relevant source code file stays the same. However, the MAP slightly decreases, meaning other but the first relevant source code files are shifted downward the ranking, i. e., are displaced by false positives, because $|S_{mod}|$ is larger in this setup as for "TraceScore Baseline". In both filtering scenarios, TraceScore is still more effective in terms of MAP and MRR than SimiScore.

> Finding 3 (RQ-3). History filtering effects TraceScore performance, being more sensitive to time interval than to the amount of source code files.

Table 4: Achieved accuracy values for AmaLgam and ABLoTS for the last 20% of all bug reports. The results for ABLoTS are averages from 10 independent classifier runs.

Project	#Bug Reports	Algorithm	Top-1	Top-5	Top-10	MAP	MRR
Axis2	216	AmaLgam ABLoTS	0.264 **0.557**	0.551 **0.658**	0.625 **0.692**	0.360 **0.564**	0.396 **0.607**
Derby	356	AmaLgam ABLoTS	0.289 **0.409**	**0.632** 0.564	**0.758** 0.622	0.369 **0.399**	0.436 **0.484**
Drools	257	AmaLgam ABLoTS	0.152 **0.499**	0.370 **0.609**	0.463 **0.654**	0.223 **0.450**	0.257 **0.553**
Hadoop	150	AmaLgam ABLoTS	0.227 **0.514**	0.580 **0.658**	0.740 **0.773**	0.350 **0.535**	0.392 **0.585**
HornetQ	54	AmaLgam ABLoTS	0.463 **0.502**	**0.741** 0.572	**0.870** 0.606	**0.502** 0.475	**0.591** 0.541
Infinispan	400	AmaLgam ABLoTS	0.150 **0.491**	0.318 **0.590**	0.443 **0.614**	0.210 **0.503**	0.239 **0.539**
Izpack	64	AmaLgam ABLoTS	0.312 **0.366**	0.500 **0.547**	**0.656** 0.584	0.368 **0.410**	0.406 **0.448**
Keycloak	158	AmaLgam ABLoTS	0.253 **0.517**	0.532 **0.609**	0.639 0.630	0.346 **0.525**	0.378 **0.565**
Log4J	89	AmaLgam ABLoTS	0.416 **0.440**	**0.787** 0.634	**0.809** 0.684	**0.520** 0.449	**0.576** 0.530
Pig	253	AmaLgam ABLoTS	0.316 **0.725**	0.680 **0.838**	0.791 **0.864**	0.417 **0.725**	0.467 **0.773**
Railo	60	AmaLgam ABLoTS	0.183 **0.573**	0.483 **0.730**	0.600 **0.732**	0.287 **0.610**	0.322 **0.636**
Seam2	156	AmaLgam ABLoTS	0.256 **0.391**	0.436 **0.443**	**0.519** 0.474	0.325 **0.392**	0.340 **0.423**
Teiid	260	AmaLgam ABLoTS	0.308 **0.491**	0.581 **0.654**	0.685 **0.695**	0.370 **0.484**	0.430 **0.571**
Weld	112	AmaLgam ABLoTS	0.223 **0.456**	0.482 **0.564**	0.598 **0.610**	0.272 **0.425**	0.331 **0.503**
Wildfly	138	AmaLgam ABLoTS	0.174 **0.368**	0.413 **0.484**	**0.522** 0.507	0.259 **0.381**	0.287 **0.424**

6.4 Do bug localization algorithms benefit from TraceScore?

Table 4 compares ABLoTS (see Section 5.4.2) with AmaLgam [53]. For AmaLgam, we set parameters $a = 0.2$ and $b = 0.3$ as proposed by their authors, and in ABLoTS "TraceScore Baseline" is used, because this configuration generally performs best (see Figures 7 and 8). AmaLgam is applied to the same 20% of bug reports as ABLoTS. ABLoTS outperforms AmaLgam in terms of Top-1, which in turn leads to improved MAP and MRR values, except for projects HornetQ and Log4J2. We performed the non-parametric Mann-Whitney U Test [32], which showed that the differences are significant ($p < 0.05$) in terms of Top-1, MAP, and MRR measures. On average, ABLoTS increases Top-1 by 102%, MAP by 49.6% and MRR by 47.8%, slightly increases in Top-5 and maintaining similar Top-10 measures compared with AmaLgam.

> Finding 4 (RQ-4) TraceScore improves bug to source code recovery when incorporated into existing algorithms (i. e. AmaLgam).

Without available source code, it is challenging to integrate TraceScore in more sophisticated algorithms such as [26, 58, 59]. A comparison based on reported metrics is also difficult, because the respectively used datasets do not contain requirements artifacts.

7 THREATS TO VALIDITY

Construct Validity. The analyzed trace links were created manually by project members in all projects implying the risk that semantically incorrect trace links were created or trace links were forgotten by mistake. Cleland-Huang et al. [8] found, that many projects outside of safety critical domains do not have reliable traceability information. However, the following aspects indicate a sufficient trace link quality in the studied projects. First, all projects' quality assurance process is based on the created trace links. The projects established a manual process where changes are reviewed and tested by humans. All projects have in common that the quality of the established trace links is implicitly verified through this process. Second, the explicit change approval process in all 15 projects ensures that the four-eyes-principle is applied for each manually created trace link. Third, the openness of all projects enables anyone to participate in the project and review the created trace links. At last, assuming trace links are imperfect, our evaluation shows, TraceScore is still able to successfully utilize this information.

External Validity. We solely focused on open-source projects, since those were the only available projects to us that provided all the necessary information to conduct this study. A potential threat to external validity arises when we want to generalize our findings to a wider population that includes commercial developments. Replications of our study with closed-sourced projects are required to justify our assumption by further empirical evidence.

[5] https://issues.apache.org/jira/browse/DERBY-6705

[6] https://issues.apache.org/jira/browse/DERBY-6357

Internal Validity. The 15 studied cases were selected by the authors of this study and thus might be biased due to certain experiences or preferences. To mitigate this threat, we specified a set of case selection criteria derived from our research questions and from the studied traceability metrics. This selection strategy ensured that we selected cases, which are suitable for the studied problem.

All bug and requirement artifacts are retrieved from projects’ issue trackers and are subject to misclassification [19], i. e. a bug is actually a feature and vice versa. Kochar et al. [23] found, misclassification affects bug localization, but the effect size is negligible. Further, comparing median number of changed source files per artifact (Table 1) shows, the values for bug reports are lower than for requirements, indicating the golden set is not artificially inflated.

Another potential threat exists in the collection and preparation of the project data. To avoid especially manual bias and to ensure reproducible results, we fully automated the process of data collection and preparation. Due to the public availability of the project artifacts and the fully automated collection and analysis process, our study can be replicated and additional projects could be included to further broaden the data corpus. We carefully verified our tool that automates this process. Therefore, we validated intermediate results of the process manually and cross-checked the data for inconsistencies and contradictions.

We split the available data set for each project into 80–20% of the bug reports retaining the temporal ordering of the project. Choosing another split point may produce different evaluation results.

8 RELATED WORK

As the manual creation and maintenance of trace links is associated with high costs [18], researchers studied information retrieval based approaches to support automated trace recovery scenarios [7, 11, 21, 30, 31, 34, 38, 48]. However, automated trace recovery implies the risk that potentially incorrect trace links are created [36, 51]. To address this traceability quality problem, Panichella et al. leveraged structural artifact information to improve the correctness of the recovered traces [39]. A combination of multiple information retrieval approaches can improve the overall recovery performance [14]. The proposed technique TraceScore recovers traces from bug reports to source code. Similar to previous approaches on supplementary bug fixing [40, 57], TraceScore utilizes graphs representing structural, historical and similarity relationships among development artifacts. Besides similarity, TraceScore also incorporate existing traces defined by the projects’ developers, as well as requirement information. Contrasting the work of Panichella et al. [14], TraceScore does not require a manual trace candidate evaluation step performed by a project developer.

Dit et al. [12] summarize *textual feature location* techniques to establish a mapping between the textual description of a feature given by the developer and parts of the source code. Several IR-based techniques, such as vector space model (VSM), and LSI have been studied the create this mapping. However, TraceScore leverages explicitly created trace links ($weight = 1$) from requirements to source code and only applies IR-based text similarity, when such information is not available. Further, the feature candidate set is pruned by two parameters $\mathcal{D}_B$ and $\mathcal{D}_R$.

Finding bugs is a costly activity in software development. IR methods are frequently proposed to automated the costly bug finding activity [1, 10]. One common approach [50, 58, 61] is to build a vector space model (VSM) to represent documents [33]. In this model the documents are encoded as vectors, where the vector elements represent the weight of terms used in a document. Different weighting schemes have been proposed. To quantify similarity, the standard way is to use cosine similarity [33]. Latent semantic Indexing (LSI) is an advanced IR retrieval technique to capture the similarity between terms and abstract concepts. Poshyvanyk et al. [42] use LSI to locate features and bugs in source code. Closely related to LSI is Latent Dirichlet allocation (LDA). LDA models documents as document distribution vectors [28, 37]. However, the evaluation in [43] found, LSI and LSA are not superior to standard VSM models, which we applied in our approach TraceScore.

The performance of IR models is usually limited by the lexical gap between queries and source code [35]. Thus, researchers invested a lot of effort in bridging the lexical gap by analyzing the source code. Various studies utilize API documents as an additional source of information [4, 9, 26, 35, 58, 59]. Saha et al. [50] structure the source code to find identifiers, method- and classnames prior extracted from bug reports. Instead of processing a source code file as a whole, Wong et al. [56] dissect the files in equally sized chunks. The chunk with the highest similarity is chosen to represent the file. To assist IR retrieval models, in [55] the authors further decreased text granularity by analyzing software changes instead of source code files. If available, bug localization methods benefit from additional data next to pure texts. Therefore bug localization algorithm ABLoTS utilizes source code information and knowledge about recently modified source code files.

9 CONCLUSION

In this paper, we studied whether utilizing additional development artifacts, namely requirements, trace links, and their history can improve bug localization. The proposed approach uses a bug report as input and utilizes a traceability graph build from historical project artifacts and relations among them. Our work led to *TraceScore* — an enhanced version history component taking advantage of explicitly defined dependency and implementation trace links as well as of requirement information available in a development project. We evaluated TraceScore on 15 large scale open-source projects with more than 13,000 bug reports in total. Further, we compared our results to two state-of-the-art approaches (*SimiScore* and *CollabScore*) that are used to analyze version history data today. The evaluation results confirm that analyzing requirements and traceability information can substantially support the bug report to source code recovery process over previous approaches. Furthermore, we showed the effectiveness of TraceScore version history component by integration into a recent bug localization algorithm.

We plan to plug *TraceScore* into more existing bug localization approaches to study its interplay with there contained components.

ACKNOWLEDGMENT

We are funded by the German Ministry of Education and Research (BMBF) grants: 01IS14026A, 01IS16003B and by the Thüringer Aufbaubank (TAB) grant: 2015FE9033.

REFERENCES

[1] Ahron Abadi, Mordechai Nisenson, and Yahalomit Simionovici. 2008. A Traceability Technique for Specifications. In *The 16th IEEE International Conference on Program Comprehension, ICPC 2008, Amsterdam, The Netherlands, June 10-13, 2008*, René L. Krikhaar, Ralf Lämmel, and Chris Verhoef (Eds.). IEEE Computer Society, 103–112. https://doi.org/10.1109/ICPC.2008.30

[2] AmaLgam 2017. AmaLgam website. https://sites.google.com/site/wswshaoweiwang/. (2017).

[3] Adrian Bachmann and Abraham Bernstein. 2009. Software Process Data Quality and Characteristics: A Historical View on Open and Closed Source Projects. In *Proceedings of the Joint International and Annual ERCIM Workshops on Principles of Software Evolution (IWPSE) and Software Evolution (Evol) Workshops (IWPSE-Evol '09)*. ACM, 119–128. https://doi.org/10.1145/1595808.1595830

[4] Sushil K. Bajracharya, Joel Ossher, and Cristina V. Lopes. 2010. Leveraging Usage Similarity for Effective Retrieval of Examples in Code Repositories. In *Proceedings of the Eighteenth ACM SIGSOFT International Symposium on Foundations of Software Engineering (FSE '10)*. ACM, New York, NY, USA, 157–166. https://doi.org/10.1145/1882291.1882316

[5] Nicolas Bettenburg, Sascha Just, Adrian Schröter, Cathrin Weiss, Rahul Premraj, and Thomas Zimmermann. 2008. What makes a good bug report?. In *Proceedings of the 16th ACM SIGSOFT International Symposium on Foundations of Software Engineering, 2008, Atlanta, Georgia, USA, November 9-14, 2008*, Mary Jean Harrold and Gail C. Murphy (Eds.). ACM. https://doi.org/10.1145/1453101.1453146

[6] Silvia Breu, Rahul Premraj, Jonathan Sillito, and Thomas Zimmermann. 2010. Information needs in bug reports: improving cooperation between developers and users. In *Proceedings of the 2010 ACM Conference on Computer Supported Cooperative Work, CSCW 2010, Savannah, Georgia, USA, February 6-10, 2010*, Kori Inkpen Quinn, Carl Gutwin, and John C. Tang (Eds.). ACM, 301–310. https://doi.org/10.1145/1718918.1718973

[7] Jane Cleland-Huang, Brian Berenbach, Stephen Clark, Raffaella Settimi, and Eli Romanova. 2007. Best Practices for Automated Traceability. *Computer* 40, 6 (2007), 27–35. https://doi.org/10.1109/MC.2007.195

[8] Jane Cleland-Huang, Orlena Gotel, Jane Huffman Hayes, Patrick Mäder, and Andrea Zisman. 2014. Software traceability: trends and future directions. In *Proceedings of the on Future of Software Engineering, FOSE 2014, Hyderabad, India, May 31 - June 7, 2014*, James D. Herbsleb and Matthew B. Dwyer (Eds.). ACM, 55–69. https://doi.org/10.1145/2593882.2593891

[9] Tathagata Dasgupta, Mark Grechanik, Evan Moritz, Bogdan Dit, and Denys Poshyvanyk. 2013. Enhancing Software Traceability by Automatically Expanding Corpora with Relevant Documentation. In *2013 IEEE International Conference on Software Maintenance, Eindhoven, The Netherlands, September 22-28, 2013*. IEEE Computer Society, 320–329. https://doi.org/10.1109/ICSM.2013.43

[10] Andrea De Lucia, Massimiliano Di Penta, and Rocco Oliveto. 2011. Improving Source Code Lexicon via Traceability and Information Retrieval. *IEEE Transactions on Software Engineering* 37, 2 (March 2011). https://doi.org/10.1109/TSE.2010.89

[11] Andrea De Lucia, Fausto Fasano, and Rocco Oliveto. 2008. Traceability management for impact analysis. In *Proceedings of the Frontiers of Software Maintenance conference*. IEEE, 21–30. https://doi.org/10.1109/FOSM.2008.4659245

[12] Bogdan Dit, Meghan Revelle, Malcom Gethers, and Denys Poshyvanyk. 2013. Feature location in source code: a taxonomy and survey. *Journal of Software: Evolution and Process* 25, 1 (2013), 53–95. https://doi.org/10.1002/smr.567

[13] Olive Jean Dunn. 1964. Multiple comparisons using rank sums. *Technometrics* 6, 3 (1964), 241–252.

[14] Malcom Gethers, Rocco Oliveto, Denys Poshyvanyk, and Andrea De Lucia. 2011. On integrating orthogonal information retrieval methods to improve traceability recovery. In *Proc. of the 27th IEEE International Conference on Software Maintenance*. 133–142. https://doi.org/10.1109/ICSM.2011.6080780

[15] Git SCM 2018. Git SCM. (2018). http://www.git-scm.com.

[16] Lan Guo, Yan Ma, Bojan Cukic, and Harshinder Singh. 2004. Robust Prediction of Fault-Proneness by Random Forests. In *Proceedings of the 15th International Symposium on Software Reliability Engineering (ISSRE '04)*. IEEE Computer Society, Washington, DC, USA, 417–428. https://doi.org/10.1109/ISSRE.2004.35

[17] Mark Hall, Eibe Frank, Geoffrey Holmes, Bernhard Pfahringer, Peter Reutemann, and Ian H. Witten. 2009. The WEKA Data Mining Software: An Update. *SIGKDD Explor. Newsl.* 11, 1 (Nov. 2009), 10–18. https://doi.org/10.1145/1656274.1656278

[18] Matthias Heindl and Stefan Biffl. 2005. A case study on value-based requirements tracing. In *Proceedings of the 10th European software engineering conference*. ACM Press, 60–69. https://doi.org/10.1145/1081706.1081717

[19] Kim Herzig, Sascha Just, and Andreas Zeller. 2013. It's not a bug, it's a feature: how misclassification impacts bug prediction. In *35th International Conference on Software Engineering, ICSE '13, San Francisco, CA, USA, May 18-26, 2013*, David Notkin, Betty H. C. Cheng, and Klaus Pohl (Eds.). IEEE Computer Society, 392–401. https://doi.org/10.1109/ICSE.2013.6606585

[20] JIRA 2018. Jira Issue Tracking Software. (2018). http://www.jira.com.

[21] Ed Keenan, Adam Czauderna, Greg Leach, Jane Cleland-Huang, Yonghee Shin, Evan Moritz, Malcom Gethers, Denys Poshyvanyk, Jonathan Maletic, Jane Huffman Hayes, Alex Dekhtyar, Daria Manukian, Shervin Hossein, and Derek Hearn. 2012. TraceLab: An Experimental Workbench for Equipping Researchers to Innovate, Synthesize, and Comparatively Evaluate Traceability Solutions. In *Proc. of the 34th International Conference on Software Engineering*. 1375–1378.

[22] Sunghun Kim, Thomas Zimmermann, E. James Whitehead Jr., and Andreas Zeller. 2007. Predicting Faults from Cached History. In *29th International Conference on Software Engineering (ICSE 2007), Minneapolis, MN, USA, May 20-26, 2007*. IEEE Computer Society. https://doi.org/10.1109/ICSE.2007.66

[23] Pavneet Singh Kochhar, Yuan Tian, and David Lo. 2014. Potential biases in bug localization: do they matter?. In *ACM/IEEE International Conference on Automated Software Engineering, ASE '14, Vasteras, Sweden - September 15 - 19, 2014*, Ivica Crnkovic, Marsha Chechik, and Paul Grünbacher (Eds.). ACM, 803–814. https://doi.org/10.1145/2642937.2642997

[24] William H Kruskal and W Allen Wallis. 1952. Use of ranks in one-criterion variance analysis. *Journal of the American statistical Association* 47, 260 (1952).

[25] Hongyu Kuang, Patrick Mäder, Hao Hu, Achraf Ghabi, LiGuo Huang, Jian Lü, and Alexander Egyed. 2015. Can method data dependencies support the assessment of traceability between requirements and source code? *Journal of Software: Evolution and Process* 27, 11 (2015), 838–866. https://doi.org/10.1002/smr.1736

[26] An Ngoc Lam, Anh Tuan Nguyen, Hoan Anh Nguyen, and Tien N. Nguyen. 2015. Combining Deep Learning with Information Retrieval to Localize Buggy Files for Bug Reports (N). In *30th IEEE/ACM International Conference on Automated Software Engineering, ASE 2015, Lincoln, NE, USA, November 9-13, 2015*, Myra B. Cohen, Lars Grunske, and Michael Whalen (Eds.). IEEE Computer Society, 476–481. https://doi.org/10.1109/ASE.2015.73

[27] C Lewis and R Ou. 2011. Bug prediction at google. http://google-engtools.blogspot.sg/2011/12/bug-prediction-at-google.html. (2011).

[28] Stacy K. Lukins, Nicholas A. Kraft, and Letha H. Etzkorn. 2008. Source Code Retrieval for Bug Localization Using Latent Dirichlet Allocation. In *WCRE 2008, Proceedings of the 15th Working Conference on Reverse Engineering, Antwerp, Belgium, October 15-18, 2008*, Ahmed E. Hassan, Andy Zaidman, and Massimiliano Di Penta (Eds.). IEEE Computer Society, 155–164. https://doi.org/10.1109/WCRE.2008.33

[29] Patrick Mäder and Alexander Egyed. 2015. Do developers benefit from requirements traceability when evolving and maintaining a software system? *Empirical Software Engineering* 20, 2 (2015). https://doi.org/10.1007/s10664-014-9314-z

[30] Anas Mahmoud and Nan Niu. 2010. Using Semantics-Enabled Information Retrieval in Requirements Tracing: An Ongoing Experimental Investigation. In *Proc. of the 34th IEEE International Computer Software and Applications Conference*. 246–247. https://doi.org/10.1109/COMPSAC.2010.29

[31] Anas Mahmoud and Nan Niu. 2011. TraCter: A tool for candidate traceability link clustering. In *Proc. of the 19th IEEE International Requirements Engineering Conference*. 335–336. https://doi.org/10.1109/RE.2011.6051663

[32] Henry B Mann and Donald R Whitney. 1947. On a test of whether one of two random variables is stochastically larger than the other. *The annals of mathematical statistics* (1947), 50–60.

[33] Christopher D. Manning, Prabhakar Raghavan, and Hinrich Schütze. 2008. *Introduction to information retrieval.* Cambridge University Press, New York.

[34] A. Marcus and J.I. Maletic. 2003. Recovering documentation-to-source-code traceability links using latent semantic indexing. In *Proceedings of the 25th International Conference on Software Engineering*. IEEE, 125–135. https://doi.org/10.1109/ICSE.2003.1201194

[35] Collin McMillan, Mark Grechanik, Denys Poshyvanyk, Chen Fu, and Qing Xie. 2012. Exemplar: A Source Code Search Engine for Finding Highly Relevant Applications. *IEEE Transactions on Software Engineering* 38, 5 (Sept. 2012), 1069–1087. https://doi.org/10.1109/TSE.2011.84

[36] Thorsten Merten, Daniel Krämer, Bastian Mager, Paul Schell, Simone Bürsner, and Barbara Paech. 2016. Do Information Retrieval Algorithms for Automated Traceability Perform Effectively on Issue Tracking System Data?. In *Requirements Engineering: Foundation for Software Quality - 22nd International Working Conference, REFSQ 2016, Gothenburg, Sweden, March 14-17, 2016, Proceedings (Lecture Notes in Computer Science)*, Maya Daneva and Oscar Pastor (Eds.), Vol. 9619. Springer, 45–62. https://doi.org/10.1007/978-3-319-30282-9_4

[37] Anh Tuan Nguyen, Tung Thanh Nguyen, Jafar M. Al-Kofahi, Hung Viet Nguyen, and Tien N. Nguyen. 2011. A topic-based approach for narrowing the search space of buggy files from a bug report. In *26th IEEE/ACM International Conference on Automated Software Engineering (ASE 2011), Lawrence, KS, USA, November 6-10, 2011*, Perry Alexander, Corina S. Pasareanu, and John G. Hosking (Eds.). IEEE Computer Society, 263–272. https://doi.org/10.1109/ASE.2011.6100062

[38] Nan Niu, Tanmay Bhowmik, Hui Liu, and Zhendong Niu. 2014. Traceability-enabled refactoring for managing just-in-time requirements. In *Proc. of the 22nd IEEE International Requirements Engineering Conference*. 133–142. https://doi.org/10.1109/RE.2014.6912255

[39] A. Panichella, C. McMillan, E. Moritz, D. Palmieri, R. Oliveto, D. Poshyvanyk, and A. De Lucia. 2013. When and How Using Structural Information to Improve IR-Based Traceability Recovery. In *Proc. of the 17th European Conference on Software Maintenance and Reengineering*. IEEE. https://doi.org/10.1109/CSMR.2013.29

[40] Jihun Park, Miryung Kim, and Doo-Hwan Bae. 2014. An empirical study on reducing omission errors in practice. In *ACM/IEEE International Conference on Automated Software Engineering, ASE '14, Vasteras, Sweden - September 15 - 19,*

2014, Ivica Crnkovic, Marsha Chechik, and Paul Grünbacher (Eds.). ACM, 121–126. https://doi.org/10.1145/2642937.2642956

[41] Porter Stemmer 2006. Porter Stemmer website. http://tartarus.org/~martin/PorterStemmer/. (2006).

[42] Denys Poshyvanyk, Yann-Gael Gueheneuc, Andrian Marcus, Giuliano Antoniol, and Vaclav Rajlich. 2007. Feature Location Using Probabilistic Ranking of Methods Based on Execution Scenarios and Information Retrieval. *IEEE Trans. Softw. Eng.* 33, 6 (June 2007), 420–432. http://dx.doi.org/10.1109/TSE.2007.1016

[43] Shivani Rao and Avinash C. Kak. 2011. Retrieval from software libraries for bug localization: a comparative study of generic and composite text models. In *Proceedings of the 8th International Working Conference on Mining Software Repositories, MSR 2011 (Co-located with ICSE), Waikiki, Honolulu, HI, USA, May 21-28, 2011, Proceedings*, Arie van Deursen, Tao Xie, and Thomas Zimmermann (Eds.). ACM, 43–52. https://doi.org/10.1145/1985441.1985451

[44] Michael Rath, David Lo, and Patrick Mäder. 2018. Replication Data for: Analyzing Requirements and Traceability Information to Improve Bug Localization. https://goo.gl/ZBpTtF. (2018). https://doi.org/doi:10.7910/DVN/N5APOB

[45] Michael Rath, Patrick Rempel, and Patrick Mäder. 2017. The IlmSeven Dataset. In *25th IEEE International Requirements Engineering Conference, RE 2017, Lisbon, Portugal, September 4-8, 2017*. IEEE Computer Society, 516–519. https://doi.org/10.1109/RE.2017.18

[46] Michael Rath, Jacob Rendall, Jin L.C. Guo, and Jane Cleland-Huang Patrick Mäder. 2018. Traceability in the Wild: Automatically Augmenting Incomplete Trace Links. In *40th International Conference on Software Engineering, ICSE '18, Gothenburg, Sweden, May 27-June 3, 2018*. ACM. https://doi.org/10.1145/3180155.3180207

[47] Patrick Rempel, Patrick Mäder, and Tobias Kuschke. 2013. An empirical study on project-specific traceability strategies. In *Proceedings of the 21st IEEE International Requirements Engineering Conference*. IEEE, 195–204. https://doi.org/10.1109/RE.2013.6636719

[48] Patrick Rempel, Patrick Mäder, and Tobias Kuschke. 2013. Towards feature-aware retrieval of refinement traces. In *Proc. of the 7th International Workshop on Traceability in Emerging Forms of Software Engineering*. IEEE. https://doi.org/10.1109/TEFSE.2013.6620163

[49] Patrick Rempel, Patrick Mäder, Tobias Kuschke, and Ilka Philippow. 2013. Requirements Traceability across Organizational Boundaries - A Survey and Taxonomy. In *Requirements Engineering: Foundation for Software Quality*, Joerg Doerr and Andreas L. Opdahl (Eds.). Vol. 7830. Springer.

[50] Ripon K. Saha, Matthew Lease, Sarfraz Khurshid, and Dewayne E. Perry. 2013. Improving bug localization using structured information retrieval. In *2013 28th IEEE/ACM International Conference on Automated Software Engineering, ASE 2013, Silicon Valley, CA, USA, November 11-15, 2013*, Ewen Denney, Tevfik Bultan, and Andreas Zeller (Eds.). IEEE, 345–355. https://doi.org/10.1109/ASE.2013.6693093

[51] A. von Knethen, B. Paech, F. Kiedaisch, and F. Houdek. 2002. Systematic requirements recycling through abstraction and traceability. In *Proceedings of the IEEE Joint International Conference on Requirements Engineering*. IEEE, 273–281. https://doi.org/10.1109/ICRE.2002.1048538

[52] Ellen M. Voorhees. 1999. The TREC-8 Question Answering Track Report. In *In Proceedings of TREC-8*. 77–82.

[53] Shaowei Wang and David Lo. 2014. Version history, similar report, and structure: putting them together for improved bug localization. In *22nd International Conference on Program Comprehension, ICPC 2014, Hyderabad, India, June 2-3, 2014*, Chanchal K. Roy, Andrew Begel, and Leon Moonen (Eds.). ACM, 53–63. https://doi.org/10.1145/2597008.2597148

[54] Shaowei Wang and David Lo. 2016. AmaLgam+: Composing Rich Information Sources for Accurate Bug Localization. *Journal of Software: Evolution and Process* 28, 10 (2016).

[55] Ming Wen, Rongxin Wu, and Shing-Chi Cheung. [n. d.]. Locus: Locating Bugs from Software Changes. In *Automated Software Engineering (ASE), 2016 31st IEEE/ACM International Conference On* (2016). IEEE, 262–273.

[56] Chu-Pan Wong, Yingfei Xiong, Hongyu Zhang, Dan Hao, Lu Zhang, and Hong Mei. [n. d.]. Boosting Bug-Report-Oriented Fault Localization with Segmentation and Stack-Trace Analysis. IEEE, 181–190. https://doi.org/10.1109/ICSME.2014.40

[57] Xin Xia and David Lo. 2017. An effective change recommendation approach for supplementary bug fixes. *Autom. Softw. Eng.* 24, 2 (2017), 455–498. https://doi.org/10.1007/s10515-016-0204-z

[58] Xin Ye, Razvan C. Bunescu, and Chang Liu. 2014. Learning to rank relevant files for bug reports using domain knowledge. In *Proceedings of the 22nd ACM SIGSOFT Int. Symposium on Foundations of Software Engineering, (FSE-22), Hong Kong, China, November 16 - 22, 2014*, Shing-Chi Cheung, Alessandro Orso, and Margaret-Anne D. Storey (Eds.). ACM. https://doi.org/10.1145/2635868.2635874

[59] Xin Ye, Hui Shen, Xiao Ma, Razvan C. Bunescu, and Chang Liu. 2016. From word embeddings to document similarities for improved information retrieval in software engineering. In *Proceedings of the 38th International Conference on Software Engineering, ICSE 2016, Austin, TX, USA, May 14-22, 2016*, Laura K. Dillon, Willem Visser, and Laurie Williams (Eds.). ACM, 404–415. https://doi.org/10.1145/2884781.2884862

[60] P. Zave, M. Jackson, E.L. Gunter, and C.A. Gunter. May-June/2000. A reference model for requirements and specifications. *IEEE Software* 17, 3 (May-June/2000), 37–43. https://doi.org/10.1109/52.896248

[61] Jian Zhou, Hongyu Zhang, and David Lo. 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 2012, June 2-9, 2012, Zurich, Switzerland*, Martin Glinz, Gail C. Murphy, and Mauro Pezzè (Eds.). IEEE Computer Society, 14–24. https://doi.org/10.1109/ICSE.2012.6227210