3-12-2022

Efficient Information Retrieval for Software Bug Localization

Saket Khatiwada
Louisiana State University and Agricultural and Mechanical College

EFFICIENT INFORMATION RETRIEVAL FOR SOFTWARE BUG LOCALIZATION

A Dissertation

Submitted to the Graduate Faculty of the
Louisiana State University and
Agricultural and Mechanical College
in partial fulfillment of the
requirements for the degree of
Doctor of Philosophy

in

Department of Computer Science

by
Saket Khatiwada
B.S. Computer Science, Southeastern Louisiana University, 2014
May 2022

To my family.

Acknowledgments

First and foremost, I would like to express my sincere gratitude to my advisor Dr. Anas Mahmoud for his continuous support during my Ph.D. study, for his patience, motivation, dedication, and immense knowledge. His guidance helped me in all the research, experiments, and writing of this thesis. I could not have imagined having a better advisor and mentor for my Ph.D. study.

Besides my advisor, I would like to thank my dissertation committee members, Dr. Doris Carver, Dr. Gerald Baumgartner, and Dr. Gabriele Piccoli, for their insightful comments, encouragement, and questions that shaped my research.

I thank my fellow doctoral students, Nishant Jha, Miroslav Tushev, and Grant Williams, for their insightful feedback, cooperation, and help throughout the years. I sincerely appreciate all the time spent on discussions, encouraging, and helping me meet my deadlines. It was fantastic to have the opportunity to work with such wonderful people. Also, thank you for all your support to all my dear friends. I was lucky to have found you guys. You know I will always cherish all the hard times we went together, the cheers, the celebrations, and the accomplishments.

Last but not least, I would like to thank my family, without whose unconditional love and support, my journey at LSU would not have been possible. I want to thank my wife Apu (Suju) Nepal and my parents, Menuka Khatiwada and Bhagawan Khatiwada, for believing in me and supporting me every step of the way. This accomplishment is as much yours as mine or even more so. Thanks for all the encouragement.

Table of Contents

List of Tables

List of Figures

Abstract

Software systems are often shipped with defects. When a bug is reported, developers use the information available in the associated report to locate source code fragments that need to be modified to fix the bug. However, as software systems evolve in size and complexity, bug localization can become a tedious and time-consuming process. Contemporary bug localization tools utilize Information Retrieval (IR) methods for automated support to minimize the manual effort. IR methods exploit the textual content of bug reports to capture and rank relevant buggy source files. However, for an IR-based bug localization tool to be useful, it must achieve adequate retrieval accuracy. Lower precision and recall can leave developers with large amounts of incorrect information to wade through. Motivated by these observations, in this dissertation, we propose a new paradigm of information-theoretic IR methods to support bug localization tasks in software systems. These methods exploit the co-occurrence patterns of code terms in software systems to reveal latent semantic information that other methods often fail to capture. We further investigate the impact of combining various IR methods on the retrieval accuracy of bug localization engines. The main assumption is that different IR methods, targeting different dimensions of similarity between software artifacts, can enhance the confidence in each other's results. Furthermore, we propose a novel approach for enhancing the performance of IR-enabled bug localization methods in the context of Open-Source Software (OSS). The proposed approach exploits knowledge from previously resolved bugs to help localize new bugs. Our analysis uses multiple datasets generated for multiple open-source and closed source projects. Our results show that *a)* information-theoretic IR methods can significantly outperform classical IR methods in bug localization tasks, *b)* optimized IR-hybrids can significantly outperform individual IR methods, and near-optimal global configurations can be determined for different combinations of IR methods, and *c)* information extracted from previously resolved bug reports can significantly enhance the accuracy of IR-enabled bug localization methods in OSS.

Chapter 1
Introduction

Bug localization is a software engineering activity concerned with finding buggy source code segments relevant to a bug report [98, 76, 100]. Despite the substantial resources devoted to delivering high-quality software, software systems are still shipped with defects. For a relatively large system, defects may range from hundreds to thousands. For example, in 2009, 4,414 bugs were reported for the Eclipse project [137]. Once a bug is reported, developers resort to the description of the bug, available in the bug report to locate source code segments that should be modified to fix the bug.

Bug localization tasks involve examining an error trace in the software system to understand the cause and isolate its relevant buggy code fragments [8]. While such a process can be feasible in smaller systems, analyzing and comprehending relatively large and complex systems can be tedious and error-prone. This problem can be particularly challenging in open-source environments, where projects can have multiple active branches maintained by distributed developer teams [119]. It has been observed that up to 70% of developer time during maintenance is devoted to program comprehension [108]. To minimize this effort, researchers employed many automated methods for software bug localization. The main objective is to partially alleviate the manual effort by helping developers automatically localize bugs and deal with the growing complexity of source code without compromising the quality of their work.

Automated methods for localizing bugs in software systems can be classified into two main categories: dynamic and static. The dynamic approach locates bugs by collecting and analyzing program data, breakpoints, or the execution traces of the system [74]. This approach relies on finding differences between the system's control flows of passing and failing runs under certain input conditions [134, 102, 136]. Dynamic methods require executing the software, which cannot always be feasible, especially in major errors that extend over several code modules. On the other hand, the static approach employs Information Retrieval

(IR) methods to locate faulty code artifacts automatically. The main assumption is that developers define their identifiers and write their comments in such a way that captures their understanding of the system at the most primitive level [4, 37, 42]. A word in this vocabulary can refer to a domain concept, a system feature, a design decision, or describe an event, an exception, or an attribute of the system [54, 69, 99]. Conventional IR methods can exploit such information to establish association relations between bug reports that are often expressed in free language and code fragments. The underlying assumption is that code fragments that share some textual similarity with a bug's description are likely related to the bug [76, 98, 100].

IR-based bug localization methods are appealing for their low computational cost and their programming language independence. Furthermore, such methods do not require a working system as they can be applied at any stage of development [12, 76, 84, 104, 137]. However, due to their inherent sub-optimal accuracy, IR methods are still far from achieving performance levels that can be adequate for practical application. Consequently, developers have to vet the outcomes of these tools to locate relevant files (true positives) and discard incorrect matches (false positives). Furthermore, recent research has shown that developers perceive an automated debugging tool as useless if it does not locate the bug's root cause early in the ranked list [10, 71, 93]. Our literature review on software bug localization has exposed several limitations in the current state of research and practice. In what follows, we review and discuss crucial related work in this domain, discuss the limitations of current bug localization techniques, and present our contributions.

1.1 Related Work

The research in IR-based bug localization has drastically progressed in recent years. Lukins et al. [76] presented an LDA-based static technique for automating bug localization tasks. The effectiveness of the proposed approach was evaluated using three case studies, including Mozilla, Eclipse, and Rhino software systems. The authors manually formulated queries from each bug report by utilizing information about the bug extracted from its title

and description in the bug report. The results showed that LDA performs as well as LSI for all bugs, achieving significantly better performance for most bugs.

Nguyen et al. [90] observed similar results using *BugScout*, a tool that uses LDA to locate buggy source code. *BugScout* considers past bug reports, in addition to source code comments and identifiers, to identify technical topics representing a source code document. Evaluating *BugScout* over four different projects showed that it correctly recommended at least one buggy file in 45% of the cases with a ranked list of 10 files.

Zhou et al. [137] proposed *BugLocator*, a tool that ranks all files of a software repository using textual similarity between bug reports and source code. The model ranks larger files higher and computes term frequency as a logarithmic function of several terms. During the ranking process, information about similar bugs that have been fixed before was considered. The tool was evaluated using 3,000 bugs collected from four open-source projects. The results showed that *BugLocator* consistently outperformed the models employing VSM, LDA, Smoothed Unigram Model (SUM), and LSI.

Saha et al. [104] introduced *BLUiR*, a tool for locating relevant files for fixing bugs. *BLUiR* ranks all the files in a software repository using the textual similarity between source code and bug reports. The textual similarity is calculated using a variant of TF.IDF that is based on the Okapi-BM25 model [103]. BM25 is a bag-of-words retrieval function. BM25 ranks documents according to their relevance to the query, regardless of the relationship between query terms within a document. *BLUiR* also adopts structured information retrieval of code constructs such as method and class names. A dataset containing 3,379 bug reports from four open-source projects was used to evaluate *BLUiR*. The results showed that using structured retrieval and exact identifier names increased the overall bug localization accuracy in terms of Top Rank (TR) 1, 5, and 10.

Thomas et al. [115] investigated the performance of a large space of IR configurations (e.g., code pre-processing, similarity metrics, and term weights) in bug localization tasks. The authors reported that different parameter settings have a considerable impact on the

method's accuracy. The analysis of 3,172 different configurations, each evaluated on 8,084 bug reports, revealed that VSM achieves the best overall performance, followed by LSI. The results also suggested that combining multiple IR methods improves performance.

Rao and Kak [100] performed a comparative study over several IR methods for bug localization, including VSM, LSI, LDA, and UM. These methods were analyzed over the iBUGS dataset [31]. The results showed that simpler text models such as UM and VSM were more effective at bug localization than more complicated models such as LDA.

Wang and Lo [121] examined the performance of various VSM variants in bug localization tasks. The authors proposed a genetic algorithm (GA) based approach to explore the space of possible compositions of different TF-IDF weighting schemes to generate a near-optimal composite model of VSM. The proposed approach was evaluated against several baselines on thousands of bug reports from AspectJ, Eclipse, and SWT. Results showed that the proposed approach improved the TR_5, MAP, and MRR rates over multiple datasets.

Chaparro et al. [21] designed and evaluated an automated approach for detecting the absence (or presence) of descriptions of Expected Behavior (EB) and Steps to Reproduce (S2R) in bug reports. This approach is intended to improve the quality of bug descriptions by alerting reporters about missing EB and S2R at reporting time. The authors utilized regular expressions, textual heuristics, and machine learning to identify EB and S2R information. These methods were evaluated on bug reports collected from 9 software projects used for open-source coding. The results showed that they were able to detect missing EB (S2R) with 85.9% (69.2%) average precision and 93.2% (83%) average recall.

Ye et al. [130] leveraged domain knowledge to bridge the lexical gap between code files and bug reports. The proposed approach ranks the source code file as a weighted combination of features that encode domain knowledge (e.g., the project's API documentation, change-history of source code). The weights were trained on a training set of bug reports using a learning-to-rank technique. The proposed approach was evaluated on six large-scale open-source Java projects. The results showed that the learning-to-rank approach significantly

outperformed state-of-the-art systems such as BugLocator [137] and BugScout [90] in recommending relevant files for bug reports.

Moreno et al. [88] used stack traces submitted in the bug report to compute the similarity between their code elements and the system's source code. The main assumption is that most stack traces available in bug reports contain code fragments related to the bug. Such information can enhance the accuracy of retrieval methods in bug localization tasks. Over 155 bug reports containing stack traces from 14 open-source systems, the proposed approach was evaluated. The results showed an improvement in performance.

Beard et al. [10] conducted a case study to determine whether the *first relevant method* (FRM) assumption is a reliable measure of the effectiveness of IR methods. In other words, does finding a single bug-relevant segment enable developers to locate all code segments relevant to the bug? The results showed that FRMs could be effective starting points for locating further relevant segments.

Le et al. [71] built an oracle to predict whether a ranked list produced by an IR-based bug localization tool is likely to contain a relevant buggy file or not. The proposed approach utilizes multiple categories of features extracted from bug reports, such as textual features and meta-data features. The authors built a separate prediction model, using Support Vector Machines (SVM), for each category of features and combined them to create a composite prediction model used as the oracle. The proposed oracle was used to evaluate the effectiveness of 3 state-of-the-art IR-based bug localization tools on more than three thousand bug reports sampled from different open-source systems. The results showed that the oracle could predict methods' effectiveness with decent levels of accuracy.

Le et al. [70] proposed a multi-modal technique that considers bug reports and program spectra (a record of which program elements are executed for each test case) to localize bugs. The proposed technique creates a bug-specific model to map a particular bug to its possible location and automatically highlight suspicious words highly associated with a bug. The *suspiciousness* score of a word is computed by considering the program elements that

contain the word. The authors evaluated their approach over 57 actual bugs from 4 software systems. The results showed that the proposed approach could outperform several baselines in terms of the number of bugs successfully localized.

Binkley et al. [13] investigated the impact of using Learning to Rank (LtR) in IR-based software maintenance tasks. LtR is a machine learning technique developed to learn how to better rank the documents retrieved using IR methods. Such a technique can learn to optimize the weights of different IR features for better accuracy. Such features included different search configurations such as keeping/removing source words, variations in splitting source code terms, and using different IR methods (VSM, LSI, and the Query Likelihood Models [96]). Evaluating LtR over two common software maintenance tasks (feature location and traceability) showed that LtR enabled statistically significant improvements in multiple performance indicators over several baseline methods.

Gethers et al. [50] used the λ-optimized approach to combine orthogonal IR techniques, which have been statistically shown to produce different results, to enhance IR-based requirements traceability tasks. In addition, the authors experimented with combining VSM, JMS, and Relational Topic Modeling (RTM) [20] on six software systems. Results indicated that the integrated method outperformed stand-alone IR and non-orthogonal combination methods with a statistically significant margin.

Crowston and Scozzi [30] studied bug fixing practices in OSS development. The authors analyzed development teams' structure and coordination practices during bug fixing practices in free/libre open-source software (FLOSS). The authors observed that the tasks for bug fixing in OSS were mostly sequential and composed of a few steps, namely submit, fix, and close. Also, bug fixing processes in OSS lacked traditional coordination mechanisms, and the effort was not equally distributed among process actors.

Anvik et al. [6] conducted a statistical study on the bug reports from open bug repositories. The authors provided characterizations of the data in bug repositories for OSS for two open-source projects, Eclipse and Firefox. The study confirmed two significant problems arising

from open bug repositories: the difficulty in detecting duplicate bug reports and assigning new bug reports to the appropriate developer.

Valdivia et al. [118] studied blocking bugs in open-source projects and proposed a model to predict whether the bugs were blocking bugs or non-blocking bugs. The authors studied bug reports from 8 open-source repositories to characterize blocking bugs using Decision Tree, Naive–Bayes, k-nearest neighbor, Zero-R, Logistic regression, Stacked Generalization, and Random Forest models. Results showed that using Decision Trees based on the C4.5 algorithm as a prediction model achieved 13.7% to 45.8% precision and 52.9% to 66.7% recall in classifying blocking and non-blocking bugs.

Wang and Kagdi [119] examined the factors that influence the likelihood of a bug getting fixed or not. Four open-source systems were used in the analysis. The authors observed that the reporters' reputations, the reputations of the developers assigned to fix the bug, and the number of comments on the bug have the most substantial impact on its probability of getting fixed. Results showed that the predictive model using features such as bug reporter's and the first assignee's reputation, attachment, and severity variables was able to achieve precision and recall of about 60% to 70%.

Open-source software projects have received enormous attention from the industry and the research community in the last several years. For example, Tantithamthavorn et al. [112] implemented bug localization using the co-change history of the OSS projects. A co-change event consists of all classes whose changes have been committed simultaneously by the same author. This concept was introduced by Ball et al. [7]. The authors experimented with bug localization over two OSS datasets, ZXing and Eclipse SWT. The results showed that taking co-change histories into account improved bug localization performance in both datasets compared to the modern bug localization tools.

Ciancarini et al. [24] developed a prediction model to differentiate concurrency bug reports from other kinds. The model was analyzed over two open-source projects, Apache HTTP Server and MariaDB. The authors used bug names, descriptions, and metadata to

classify bug reports with an average precision and recall of 0.8785 and 0.729, respectively. Furthermore, the authors reported that using linked bug information (bug reports information in the bug repository and the corresponding fix in the version control system) for the classification resulted in average precision and recall of 0.8995 and 0.7525, respectively.

Wu et al. [127] proposed ChangeLocator to automatically locate crash-inducing changes for a given bucket of crash reports. The approach was based on the learning model that used features from the study and trained it using data from historically fixed crashes. The authors evaluated ChangeLocator using six release versions of the Netbeans project, an open-source project. The results showed that ChangeLocator could locate the crash-inducing changes for 44.7%, 68.5%, and 74.5% of the bugs by examining the top 1, 5, and 10 changes in the recommended list, respectively. Thus, ChangeLocator significantly outperformed the existing state-of-the-art approaches.

Wang and Lo [122] proposed AmaLgam, a method for locating relevant buggy files using version history, similar reports, and structure in addition to bug reports to localize bugs. AmaLgam integrates a bug prediction technique used in Google that analyzes version history, BugLocator [137], which analyzes similar reports from bug report system, and BLUiR [104], which considers structure. The method was experimented on four open-source projects, namely AspectJ, Eclipse, SWT, and ZXing, to localize more than 3,000 bugs. Results showed that the method outperformed the history-aware bug localization solution [107], BugLocator, and BLUiR by 46.1%, 24.4%, and 16.4%, respectively.

Mills et al. [87] studied the effectiveness of Text Retrieval (TR) in localizing bugs. The experiment was conducted on 620 bug reports manually extracted from verified 13 software systems. The authors analyzed bug report texts with and without localization hints (e.g., code snippets, method names, etc.) using a genetic algorithm to derive a near-optimal query that provides insight into the potential of that bug report for using TR-based localization. The authors found strong evidence that bug reports themselves provide sufficient information to perform bug localization.

Cabot et al. [19] analyzed over three million Github projects to give some insights on how labels are used in such projects. The authors analyzed more than three million GitHub projects on how labels are used in OSS projects. Results showed that, although labels are scarcely used in the projects, using labels favored the resolution of the issues. Results also showed that the labels indicate bugs, enhancements, priorities, etc.

Youm et al. [132] proposed Bug Localization with Integrated Analysis (BLIA) for a statically integrated bug localization by utilizing texts and stack traces in bug reports, structured information of source files, and source code change histories. The authors experimented with varying parameters to find each score's optimized combination and impact on bug localization performance. Three open-source projects, namely AspectJ, SWT, and ZXing, were used to measure the performance of BLIA. Results showed that using optimal parameters, BLIA outperformed the existing tools, namely BugLocator, BLUiR, BRTracer, and AmaLgam, by 34%, 23%, 17%, and 8%, respectively in terms of mean average precision.

Vasquez et al. [73] proposed an expert developers recommendation approach using a combination of an information retrieval technique and processing of the source code authorship information. The approach was evaluated on three open-source software, rgoUML, JEdit, and MuCommander. Compared with approaches that *i)* uses machine learning or *ii)* data mining of source code, the authors reported the recommendation accuracy of the proposed approach is equivalent or better than the two compared approaches.

1.2 Contributions

Our brief review of seminal related work shows that the lion's share of current work is focused on investigating the performance of various IR methods in matching bug reports with their related buggy code artifacts (e.g. [100, 115, 76, 64]). Other lines of research include the design, development, and evaluation of tools based on these IR methods (e.g. [104, 137, 90]), techniques for enhancing the performance of existing IR methods by exploiting different sources of information in bug reports and code artifacts (e.g. [21, 130, 88]), or study focused on the quality of measures used to assess the effectiveness of IR-based methods and tools

(e.g. [10, 71]). Our work builds upon existing work to propose a more effective solution for the problem. In particular, to advance state of the art, in this dissertation, we:

- Propose a new paradigm of information-theoretic IR methods to support bug localization tasks in software systems. The proposed methods include Pointwise Mutual Information (PMI) and Normalized Google Distance (NGD).
- Investigate the impact of combining various IR methods into hybrid pairs on the performance of code retrieval engines. The main assumption is that different IR methods can target different dimensions of similarity between artifacts, thus, enhancing the overall performance of IR-enabled bug localization methods. We also present near-optimal configurations for combining several IR methods into hybrid pairs.
- Investigate the effectiveness of our proposed information-theoretic IR methods in localizing bugs in Open Source Software (OSS). We further propose a systematic process for preparing ground-truth BL datasets for OSS projects and introduce a novel method for enhancing the retrieval accuracy of IR methods using information extracted from previously resolved bug reports.

Chapter 2
An Information-Theoretic IR Approach for BL

IR methods exploit the textual content of bug reports to capture and rank relevant buggy source files. In this chapter, we propose a new paradigm of information-theoretic IR methods to support bug localization tasks in software systems. These methods, including Pointwise Mutual Information (PMI) and Normalized Google Distance (NGD), exploit the co-occurrence patterns of code terms in the software system to reveal hidden textual semantic dimensions that other methods often fail to capture. Our objective is to establish accurate semantic similarity relations between source code and bug reports. The main assumption is that methods that consider the contextual usage of terms in the system can uncover semantic dimensions that other methods often fail to capture. The proposed methods are compared against classical IR methods commonly used in bug localization research. Five benchmark datasets from different application domains are used to conduct our analysis. The results show that information-theoretic co-occurrence methods provide just enough semantics necessary to establish relations between bug reports and code artifacts, achieving a balance between simple lexical methods and computationally-expensive semantic IR methods that require substantial amounts of data to function properly.

2.1 Introduction

A broad range of IR methods, at various levels of computational complexity, have been investigated in bug localization research. Such methods range from simple lexical matching, such as Vector Space Model (VSM) [105], to more computationally-expensive semantic methods such as Latent Semantic Indexing (LSI) [36] and Latent Dirichlet Allocation (LDA) [14]. Despite these advances, the applications of conventional IR methods in bug localization are frequently sub-optimal [116]. Problems arise because source code employs a restricted vocabulary that lacks uniqueness and exhibits a high degree of repetition [58, 48]. Even code comments, supposedly written in free natural language, tend to gradually narrow into a lexically and semantically restricted subset of English [44]. This leads data extensive IR

methods, such as LDA and LSI, to poorly perform [34, 78]. Furthermore, as projects evolve, new and inconsistent terminology gradually finds its way into the system [2, 79], causing related system artifacts to exhibit a high degree of variance in their contents [4, 78, 45]. This phenomenon, known as the *vocabulary mismatch* problem, is regarded as one of the principal causes of declining accuracy in retrieval engines [34, 78]. As a result, IR methods applied to bug localization tasks are still far from achieving accuracy levels adequate for practical applications. Such methods cannot achieve high recall without retrieving many false positives. Low levels of accuracy leave developers with a considerable amount of manual effort to vet the outcome of these methods.

Motivated by these observations, in this chapter, we investigate the performance of a new paradigm of information-theoretic IR methods that exploit shallow semantic attributes of the textual content of software artifacts. The proposed methods, including Pointwise Mutual Information (PMI) [23] and Normalized Google Distance (NGD) [25], exploit the contextual cues of terms in a software system, embedded in their co-occurrence patterns, to estimate textual semantic similarity between source code artifacts and bug reports. A preliminary assumption is that co-occurrence-based methods can capture a system-wide semantic dimension that other IR methods often fail to capture, thus, achieving a balance between lexical methods that rely solely on text matching, and computationally expensive semantic methods that rely on analyzing the latent semantic structures of software artifacts.

Five benchmark bug localization datasets are used to conduct our analysis in this paper. In particular, we compare the performance of the proposed methods against a series of IR methods that have been extensively used in bug localization research. These methods include Vector Space Model (VSM) [105], as a representative of lexical matching methods, Latent Semantic Indexing (LSI) [36], as a representative of semantically enabled IR methods, and Jensen-Shannon Model (JSM) [1], as a representative of probabilistic IR methods. The performance in our analysis is assessed using a series of performance measures designed to capture various accuracy and browsability aspects of the proposed methods. In what follows,

we introduce the information-theoretic IR methods, describe our experimental setup, present our results, and discuss our main findings.

2.2 Information-Theoretic IR

Conventional Information Retrieval (IR) methods have been recently applied to retrieve software artifacts. Such methods aim to match a query of keywords with a set of artifacts in a software repository. The retrieved artifacts are typically ranked in descending order according to their relevance to the query using a predefined similarity measure. IR methods have been leveraged to provide automated support for several basic software engineering tasks, such as, mining software repositories [114], concept location [98, 85, 26], software reuse [46, 77], source code refactoring [111, 9], code summarization [53], software traceability [79, 5, 83, 56], and bug localization [76, 104, 90].

In this section, we describe an information-theoretic IR approach for matching bug reports with code artifacts. The proposed approach is based on the fact that the accuracy of text retrieval methods relies on the accuracy of similarity relations established between individual words in the corpus [72, 82]. Two methods of semantic similarity are investigated in our analysis. These methods include Pointwise Mutual Information (PMI) [23] and Normalized Google Distance (NGD) [25]. PMI and NGD exploit the distributional cues of terms in the system to estimate their pairwise semantic relatedness. The main assumption is that the statistical co-occurrence of words reflects their similarity status. Words that frequently appear together in the same context are highly likely to be related. In NLP research, relatedness is a broader concept of similarity, accounting for relations between words that are considered related even though they are not similar from a linguistic point of view. More specifically, the similarity is typically estimated by considering the lexical relations of synonymy, or equivalent words, and hypernyms, or type-of relation, between words. On the other hand, relatedness is estimated by considering all contextual relations between two words, including their statistical co-occurrence relations [3, 17, 49, 110]. Therefore, similarity can be considered as a special case of relatedness. For example, in English, the words *mouse*

and *keyboard* are not considered similar (neither synonyms nor hypernyms). However, any person with a basic knowledge of computer systems would regard these two words to be *very related* since they often appear together as examples of computer peripherals.

In the context of source code, the notion of relatedness seems to be more appropriate than similarity to describe the lexical, structural, and semantic associations between code terms. For instance, two code terms could be related purely based on their functional relations. Therefore, methods such as PMI and NGD should capture more accurate association relations between terms in a software system, consequently enabling a more accurate code artifact retrieval process. Based on these assumptions, we expect that using PMI or NGD as a basis for matching bug reports and code artifacts can lead to higher retrieval accuracy. In what follows, we describe in greater detail these two methods, along with Textual Semantic Similarity (TSS) [86], an IR method that utilizes the term semantic similarity to estimate semantic relatedness between software artifacts.

2.2.1 Pointwise Mutual Information

PMI is an information-theoretic measure of information overlap, or statistical dependence, between two words [23]. Church and Hanks [23] introduced PMI and later used by Turney [117] to identify synonym pairs using Web search results. Formally, PMI between two words w_1 and w_2 can be measured as the probability of them occurring in the same text versus their probabilities of occurring separately. Assuming the collection contains N documents, PMI can be calculated as:

$$PMI = \log_2\left(\frac{\frac{C(w_1,w_2)}{N}}{\frac{C(w_1)}{N}\frac{C(w_2)}{N}}\right) = \log_2\left(\frac{P(w_1,w_2)}{P(w_1)P(w_2)}\right) \tag{2.1}$$

where $C(w_1, w_2)$ is the number of documents in the collection containing both w_1 and w_2 and $C(w_1)$, $C(w_2)$ are the numbers of documents containing w_1 and w_2, respectively. Mutual information compares the probability of observing w_1 and w_2 together against the probabilities of observing w_1 and w_2 independently. Formally, mutual information is a measure of how much the actual probability of co-occurrence of an event $P(w_1, w_2)$ differs

from the expectation based on the assumption of independence of $P(w_1)$ and $P(w_2)$ [16]. If the words w_1 and w_2 are frequently associated, the probability of observing w_1 and w_2 together will be much larger than the chance of observing them independently. This results in a PMI >> 1. On the other hand, if there is no relation between w_1 and w_2, then the probability of observing w_1 and w_2 together will be much less than the probability of observing them independently (i.e., PMI << 1).

PMI is symmetrical; the amount of information acquired about w_2 from observing w_1 is equivalent to that of information acquired about w_1 when observing w_2. The value of PMI can go from $-\infty$ to $+\infty$. A $-\infty$ value of PMI indicates that the two words are not related or do not appear together in any of the system's artifacts, and $+\infty$ indicates a complete co-occurrence between the words. The value of PMI can be normalized as follows [16]:

$$nPMI = \frac{PMI}{-\log_2(P(w_1, w_2))} = \frac{\log_2(P(w_1)P(w_2))}{\log_2(P(w_1, w_2))} - 1 \tag{2.2}$$

Given this formula, PMI can take the following values:

- PMI = -1: In this case, w_1 and w_2 do not co-occur in any document in the corpus, i.e., $P(w_1, w_2) = 0$,
- -1 < PMI < 1: In this case, the two words w_1 and w_2 do occur in several documents, i.e., $P(w_1, w_2) \neq P(w_1)P(w_2)$
- PMI = 1: This case is known as the perfect dependence case, where the w_1 and w_2 only occur together $P(w_1, w_2) = P(w_1) = P(w_2)$,

PMI is intuitive, scalable, and computationally efficient [89, 86]. These attributes make PMI an appealing similarity method to process massive corpora of textual data in tasks such as short-text retrieval [86], Semantic Web [117, 109], and text and data mining [59].

2.2.2 Normalized Google Distance

Normalized Google Distance (NGD) uses Google counts to devise a semantic relatedness measure between words based on information distance and Kolmogorov complexity [25].

The main assumption is that the statistical co-occurrence of words in the Web reflects their current similarity status in society, thus indicating their relatedness. Formally, to estimate the semantic distance between two words (w_1 and w_2) using NGD, a Google search query Q is requested for $Q(w_1)$, $Q(w_2)$, and $Q(w_1 \text{ AND } w_2)$. The semantic relatedness is then measured as:

$$NGD(w_1, w_2) = \frac{\max\{\log(D_1), \log(D_2)\} - \log(|D_1 \cap D_2|)}{\log(|D|) - \min\{\log(D_1), \log(D_2)\}} \tag{2.3}$$

where D_1 and D_2 are the hit counts of $Q(w_1)$ and $Q(w_2)$, or the number of documents that contain w_1 and w_2, respectively. $|D_1 \cap D_2|$ is the hit count of $Q(w_1 \text{ AND } w_2)$ or the number of Web documents that contain both w_1 and w_2. The main tenet is that pages that contain both terms indicate relatedness, while pages that contain only one of the terms suggest the opposite. NGD is a distance measure, meaning that the NGD of two words that occur together is 0, and the NGD of two words that do not occur together on any Web page is ∞. To bound NGD value between 0 and 1, the following formula is often used [51]:

$$nNGD = e^{-2 \cdot NGD(w_1, w_2)} \tag{2.4}$$

The theoretical derivation of NGD can be described as follows: Let a be a software artifact that contains the term x such that $x \in a$. The probability of x in a system that contains $\mathbf{M}$ number of artifacts is $p(x) = |\mathbf{x}|/\mathbf{M}$, where $\mathbf{x}$ is the set of artifacts containing x. Let $\mathbf{x} \cap \mathbf{y}$ be the number of artifacts in the system that contain both terms x and y such that $\mathbf{x} \cap \mathbf{y} = \{a : x, y \in a\}$, then, the joint probability of x and y can be calculated as $p(x, y) = |\{a : x, y \in a\}|/\mathbf{M}$ and the probability $p(x|y)$ of conditional events $\mathbf{x}|\mathbf{y} = (\mathbf{x} \cap \mathbf{y})/\mathbf{y}$ can be calculated as $p(x|y) = p(x, y)/p(y)$. For example, assuming three code terms from a software system, $\mathbf{S}$, including *int*, *userID*, and *pwd*, appear in 240, 16, and 12 artifacts, respectively. *userID* and *pwd* appear together in 10 artifacts, and *userID* and *int* appear together in 16 artifacts. Assuming $\mathbf{S}$ has a total of 250 artifacts, then:

$$p(\texttt{userID}|\texttt{pwd}) = p(\texttt{userID}, \texttt{pwd})/p(\texttt{pwd}) = 0.833 \tag{2.5}$$

Similarly, $p(\texttt{pwd}|\texttt{userID}) = 0.625$, $p(\texttt{userID}|\texttt{int}) \simeq 1$, and $p(\texttt{int}|\texttt{userID}) \ll 1$. However, since the term *int* appears in almost all the artifacts in the system, it tells us almost nothing about any other term in the system, thus it makes sense to assume that the distance between any two terms in the system x, y is best given by:

$$D_1(x, y) = min\{p(x|y), p(y|x)\} \tag{2.6}$$

However, relying on this measure gives misleading results because the difference among smaller probabilities has more significance the smaller the probabilities are. The negative logarithm of items being minimized is taken to resolve this problem, producing D_2:

$$D_2(x, y) = max\{log(1/p(x|y)), log(1/p(y|x))\} \tag{2.7}$$

Moreover, since D_1 deals with absolute probabilities, two terms with very small similarities are considered much less similar than two terms with much larger probability but the same D_1. This can lead to a problem in the context of software systems as some terms are very rare, appearing in only a few artifacts. To overcome this problem, D_2 is normalized by dividing by the maximum of $\log 1/p(x)$, $\log 1/p(y)$, producing D_3 as follows:

$$D_3(x, y) = \frac{max\{log(1/p(x|y)), log(1/p(y|x))\}}{max\{log(1/p(x)), log(1/p(y))\}} \tag{2.8}$$

Given that the frequency of x is $f(x) = \mathbf{M}p(x)$, $f(x, y) = \mathbf{M}p(x, y)$, and $p(x|y) = f(x, y)/f(y)$, D_3 can be rewritten in terms of terms frequency as follows:

$$NGD(x, y) = \frac{\max\{\log(f(x)), \log(f(y))\} - \log(|f(x) \cap f(y)|)}{\log(\mathbf{M}) - \min\{\log(f(x)), \log(f(y))\}} \tag{2.9}$$

NGD has gained momentum in recent years due to its solid theoretical foundation,

simplicity, relatively low computational complexity across several applications, and ability to achieve a decent correlation with the human perception of relatedness [78, 128, 126, 22].

2.2.3 Text Semantic Similarity

Proposed by Mihalcea et al. [86], Text Semantic Similarity (TSS) is a corpus-based measure that estimates semantic similarity between two texts using the pairwise semantic similarity of their words. TSS models the semantic similarity of texts as a function of the semantic similarity of their words. In particular, TSS combines word-to-word similarity and word specificity to calculate the similarity between paragraphs. Formally, the similarity between two texts, T_1 and T_2, is calculated as:

$$\frac{1}{2}\left(\frac{\sum(maxSim(w_i, T_2) \times IDF(w_i))}{\sum IDF(w_i)} + \frac{\sum(maxSim(w_j, T_1) \times IDF(w_j))}{\sum IDF(w_j)}\right) \tag{2.10}$$

$maxSim(w_i, T_1)$ returns the similarity score between w_i from T_1 and its most similar word in T_2. Similarly, $maxSim(w_j, T_2)$ returns the similarity score between the word w_j from the text T_2 and its most similar word in T_1. $IDF(w)$ is the word's specificity, calculated as the number of artifacts and topics in the system divided by the number of artifacts and topics that contain the word w.

TSS was originally introduced to calculate the similarity between short texts. Short-text is a relatively recent Natural Language Processing (NLP) classification of text that has been motivated by the explosive growth of micro blogs on social media (e.g., Tweets and YouTube comments), and the urgent need for effective methods to analyze such large amounts of limited textual data [52]. The main characteristics of short-text include data sparsity, noisy content, and colloquial terminologies. Such characteristics can also be observed in source code, including the limited vocabulary used, the noisy content generated by arbitrary naming conventions, and the terminologies emerging from domain-specific acronyms and abbreviations [95, 57].

Table 2.1. Co-occurrence Matrix **A**

	file	path	password	search	user
file	5	3	4	1	4
path	3	4	3	1	3
password	4	3	11	1	10
search	1	1	1	1	1
user	4	3	10	1	77

Table 2.2. Matrix ($B_{\mathbf{NGD}}$): The NGD Matrix of **A**

	file	path	password	search	user
file	1	0.73	0.51	0.50	0.14
path	0.73	1	0.45	0.55	0.13
password	0.51	0.45	1	0.35	0.16
search	0.50	0.55	0.35	1	0.15
user	0.14	0.13	0.16	0.15	1

Table 2.3. Matrix ($B_{\mathbf{PMI}}$): The PMI Matrix of **A**

	file	path	password	search	user
file	1	0.77	0.62	0.65	0.01
path	0.77	1	0.55	0.70	-0.01
password	0.62	0.55	1	0.48	0.07
search	0.65	0.70	0.48	1	0.06
user	0.01	-0.01	0.07	0.06	1

Table 2.4. Pairwise TSS Matrix ($\mathbf{TSS_{NGD}}$) with NGD

	a.java	b.java	c.java
a.java	1	0.85	0.83
b.java	0.85	1	0.91
c.java	0.83	0.91	1

Table 2.5. Pairwise TSS Matrix ($\mathbf{TSS_{PMI}}$) with PMI

	a.java	b.java	c.java
a.java	1	0.89	0.88
b.java	0.89	1	0.92
c.java	0.88	0.92	1

To demonstrate the operation of PMI, NGD, and TSS, we use the following example. ***Example:*** Matrix **A** (Table 2.1) represents a sample partial co-occurrence matrix of terms extracted from a software system with 116 artifacts. The matrix shows the co-occurrence data for the terms {`file`, `path`, `password`, `search`, `user`}. The NGD and the PMI matrices of these terms are given in Table 2.2 and 2.3, respectively. Assuming *a.java*, *b.java* and *c.java* are three code artifacts, such that:

- *a.java* = { `file`, `path`, `password` }
- *b.java* = { `path`, `password`, `search` }
- *c.java* = { `password`, `search`, `user`, `file` }

The pairwise TSS similarities between these artifacts are shown in the matrices in

Table 2.6. Bug Localization Experimental Datasets

System	Bug Reports	Source File
AspectJ [101]	318	6503
Eclipse [137]	3075	12300
JodaTime [101]	43	315
SWT [137]	98	484
ZXing [137]	20	391

Table 2.4 and 2.5. In Table 2.4, TSS (Eq. 2.10) is calculated using NGD (Eq. 2.3) as the semantic similarity measure between code terms. We refer to this measure as TSS_{NGD}. In Table 2.5, TSS is calculated using PMI (Eq. 2.10) as the semantic similarity measure between code terms. We refer to this measure as TSS_{PMI}.

2.3 Experimental Setup

In this section, we describe the benchmark datasets used to carry out our analysis, the IR methods investigated as experimental baselines, the evaluation measures used to assess the performance of the investigated methods, and our main research questions.

2.3.1 Experimental Datasets

We use five benchmark datasets commonly employed in bug localization research to conduct our analysis. Table 2.6 describes these datasets, including the number of bug reports and source files reported in each dataset. These we obtained from two different sources, including:

- moreBugs: The *moreBugs* [101] dataset consists of bug reports from the AspectJ and the JodaTime repositories. For each bug report, moreBugs includes the bug's title, its description, and the list of files modified to fix the bug. The dataset consists of 318 bug reports for the AspectJ system and 43 bug reports for the JodaTime system.

- Zhou et al. [137]: This dataset includes bug reports from three popular open-source projects: *SWT*, *ZXing*, and *Eclipse*. The dataset consists of the bug's title, its description,

and the list of files modified to fix the bug. This dataset consists of 98, 20, and 3075 bug reports for *SWT*, *ZXing*, and *Eclipse* respectively.

2.3.2 Experimental Baselines

The performance of the proposed methods (NGD, PMI) is compared against a series of IR methods that have been extensively used in bug localization research. These methods include VSM [105], as a representative of lexical matching methods, LSI [36], as a representative of semantically enabled methods, and JSM [1], as a representative of probabilistic IR methods. The following is a description of these methods in greater detail:

- **Vector Space Model**: VSM (or Term Vector Model) is an algebraic model that consists of a single term-document matrix. Each row of the matrix represents a single term or word found in the corpus, and each column represents an individual document. Each entry in the matrix $w_{i,j}$ is the weight of the term j in the document i, indicating the importance of the term to the document's subject matter. While the raw frequency of the term in the document can be used as a weight, in VSM calculations, another approach, known as term frequency - inverse document frequency (TF.IDF) is typically used [81]. TF.IDF is calculated as the product of the frequency of the term in the document (TF) and the term's scarcity across all the documents (IDF). Formally, TF.IDF can be computed as:

$$TF.IDF = f(w,d) \times \log \frac{|D|}{|d_i : w \in d_i \wedge d_i \in D|} \tag{2.11}$$

where *f(w,d)* is the frequency of the word w in document d, D is the total number of documents in the corpus, and $|d_i : w \in d_i \wedge d_i \in D|$ is the number of documents in the corpus D that contain the word w. Given this weighted vector representation of documents, the similarity between two documents in the corpus can be determined by calculating the distance between their document vectors. The cosine distance is widely used to measure the similarity between documents, however other distance measures

such as Euclidean distance, Hellinger distance, or KL divergence can also be used. For a query q and a document d, the cosine similarity is computed as:

$$Similarity(q, d) = \frac{\vec{V_q} \cdot \vec{V_d}}{|\vec{V_q}||\vec{V_d}|} \tag{2.12}$$

where V_q and V_d are the vectors of term weights for the document d and the query q respectively and $\vec{V_q} \cdot \vec{V_d}$ represents the dot product of the two vectors.

- **Latent Semantic Indexing**: LSI [36] is an IR method used to identify the relationship between the terms and concepts contained in an unstructured collection of text. LSI draws on the assumption that some underlying (latent) semantic structure is partially concealed by the variability of the contextual usage of words in a certain collection [40]. Formally, this process starts by constructing a single term-document matrix of size $m \times n$ where m is the number of terms and n is the number of documents in the corpus. Singular Value Decomposition (SVD) is then used to reduce the matrix to a product of three matrices in the form:

$$A = U\Sigma V^T \tag{2.13}$$

where A is the term-document matrix, U is $m \times n$ orthogonal matrix, Σ is a $n \times n$ matrix of singular values and V is a $n \times n$ orthonormal matrix. The key insight in LSI is to reduce the dimensionality of the information retrieval problem. This is achieved by deleting all but k largest values on the diagonal of Σ matrix and corresponding columns in the other two matrices. The resulting truncated matrices are represented as U_k, Σ_k, and V_k for U, Σ, and V, respectively. This truncation process generates k-dimensional vector space, the vectors in which represent each document. Similarly, the query vector in this k-dimensional space can be calculated as:

$$v = q^T U_k \Sigma_k^{-1} \tag{2.14}$$

where q is the *tf*-vector for the query. Similarity between the query and a document is computed by calculating the cosine distance between the computed vector v and the column vector representing the document in V_k^T matrix.

- **Jensen-Shannon Model**: Presented by Abadi et al. [1], JSM is a probabilistic IR techniques that represents each document d in the corpus as a probabilistic distribution over its words, such that:

$$p_d = \frac{f(w,d)}{T_d} \tag{2.15}$$

In this formula, $f(w,d)$ is the frequency of the word w in document d, and T_d is the total number of words in d. The similarity between the two documents can be estimated using the distance between their probabilistic distributions calculated using the Jensen-Shannon Divergence [28] as:

$$JSM(d,q) = 1 - JS(p_d, p_q) \tag{2.16}$$

The Jensen-Shannon Divergence is the average information-theoretic Kullback-Leibler divergence [66] of each of two distributions to their average distribution, such that:

$$JSM(d,q) = 1 - \left[H(\frac{p_d + p_q}{2}) - \frac{H(p_d) + H(p_q)}{2} \right] \tag{2.17}$$

where p_d and p_q are the empirical distributions of the documents d and q, respectively, and H is the entropy of the distribution p given by:

$$H(p) = \sum_{j=1}^{n} p(x_j) \cdot \log_2 p(x_j) \tag{2.18}$$

JSM value can range from [0, 1], where 1 indicates a perfect similarity. JSM has been used in software traceability tasks, and it has been shown to outperform methods such as VSM and LSI [91, 1].

2.3.3 Evaluation Metrics

In our analysis, we deal with a classical IR problem. A standard text retrieval process includes an IR method matching a query of keywords with a collection of documents in a corpus. The retrieved documents are ranked in a list format according to their similarity to the query. This list contains relevant (true positives) and irrelevant (false positive) links. Ideally, an effective IR method should place relevant links at the top of the list. Based on these definitions, in what follows, we describe the set of measures we use to evaluate and compare the performance of the different IR methods used in our analysis:

- **Precision and Recall:** Precision and recall are the most popular measures of IR effectiveness. Recall is a measure of coverage or the percentage of correct links that are retrieved. Precision is a measure of accuracy, or the percentage of retrieved links that are correct. Formally, if A is the set of correct links and B is the set of retrieved links, then recall ($\boldsymbol{R}$) and precision ($\boldsymbol{P}$) can be calculated as:

$$R = |A \cap B|/|A| \tag{2.19}$$

$$P = |A \cap B|/|B| \tag{2.20}$$

- **Mean Average Precision (MAP):** MAP is often used in IR research to assess the browsability of the results. Browsability in our analysis can be defined as the extent to which the organization of a list, generated by the automated relatedness method,

eases the effort for the analyst to find actual related source files. The main assumption is that for an automated method to be considered effective, it has to place related files at the upper part of the ranked list.

Formally, MAP is a measure of quality across recall levels [81]. For each query, a cutoff point is taken after each true link in the ranked list of candidate links. The precision is then calculated. Correct links that were not retrieved (false negatives) are given a precision of 0. The precision values for each query are then averaged over all the relevant links (true positives) in the answer set of that query ($|C|$), producing Average Precision (AP). The Mean Average Precision (MAP) is calculated as the average of AP for all queries in each dataset [113]. MAP indicates the order in which the returned documents are presented. For instance, if two IR methods retrieved the same number of correct links (same recall), the method that places more correct links toward the top of the list will have a higher MAP. Eq. 2.21 describes MAP, assuming the dataset has Q bug reports.

$$MAP = \frac{1}{|Q|} \sum_{j=1}^{|Q|} \frac{1}{|C_j|} \sum_{i=1}^{|M_j|} Precision(M_j, i) \tag{2.21}$$

In this equation, $Precision(M_j, i)$ is the precision of the generated list M_j after a cut-off point is taken at the level at which the correct link (true positive) i is found in the list.

- **Mean Reciprocal Rank (MRR)**: Reciprocal rank (RR) is the multiplicative inverse of the rank of the first correct item of a query. Specifically, it measures how early a correct item appears in a ranked list. This measure is used when the user is mainly concerned with retrieving at least one correct item. For instance, if the first relevant link occurs at rank n, the reciprocal rank is computed as $\frac{1}{n}$ [29]. MRR is the average of the reciprocal ranks across all queries Q. It can be calculated as:

$$MRR = \frac{1}{|Q|}\sum_{i=1}^{|Q|}\frac{1}{rank_i} \tag{2.22}$$

where $rank_i$ is the rank of the first relevant link for the query Q_i. MRR ranges between 0 and 1, where 1 represents a perfect retrieval scenario (i.e., a correct item is positioned at rank 1 for each query).

- **Top N Rank (TR_N):** This measure returns the percentage of queries that contains at least one relevant file in the first N files of the retrieved list. Different values of N can be used; in our analysis, we use 1 (TR_1), 5 (TR_5), and 10 (TR_{10}). For example, a TR_1= 50% indicates that 50% of the queries returned a relevant file as the top-ranked file in the list (first hit).

  MRR and TR_N are effective indicators of the practicality of IR-based bug localization tools. Specifically, there are very few source code artifacts related to each reported bug. The ranking of these correct artifacts is critical for the effectiveness of the IR method, as identifying at least one buggy source file often makes it easier for developers to find the rest [104]. Previous research reported that developers would perceive an automated debugging tool as not-useful if it does not locate the root cause of a bug early in the ranked list [10, 71, 93].

Example: To explain the operation of these different measures, consider the retrieval scenario in Fig 2.1.

Assume a corpus with 10 documents. An IR method M is used to retrieve the set of related artifacts for two different queries, Q_1 and Q_2. Q_1 has 4 relevant documents in the corpus, while Q_2 has only 3. Assuming M generated the two lists of candidate relevant documents (links) in Fig .2.1-a and 2.1-b for Q_1 and Q_2, respectively. Given this scenario, our different assessment measures can be calculated as follows:

- Precision and Recall: For Q_1, a total of 10 files were retrieved. All relevant files

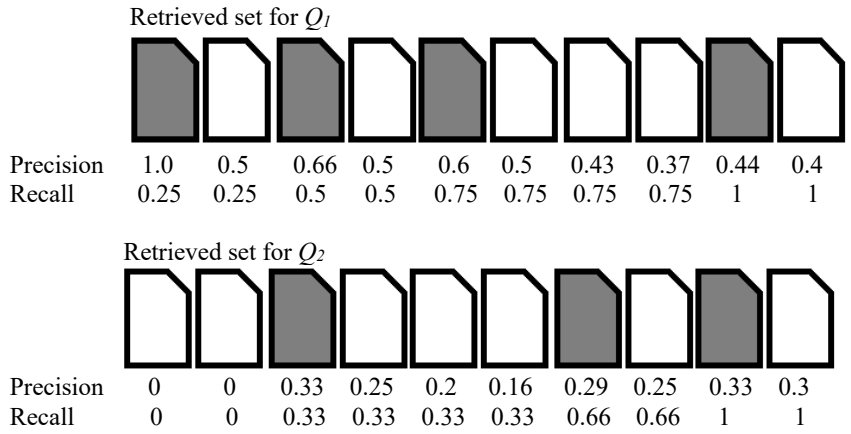


Figure 2.1. A walkthrough example: Illustration of the performance measures

appeared in the list. Therefore,

$$R_{Q_1} = \frac{4}{4} = 100\% \quad and \quad P_{Q_1} = \frac{4}{10} = 40\%$$

For Q_2, a total of 10 files were retrieved. All relevant files appeared in the list. Therefore,

$$R_{Q_2} = \frac{3}{3} = 100\% \quad and \quad P_{Q_1} = \frac{3}{10} = 30\%$$

- TR_1: For Q_1, the first link is a relevant file. However, for Q_2, the first file is irrelevant (false positive). Therefore,

$$TR_1 = \frac{1}{2} = 50\%$$

- TR_5: The first 5 files retrieved for both Q_1 and Q_2 contain at least one relevant file each. Therefore,

$$TR_5 = \frac{2}{2} = 100\%$$

- TR_{10}: The first 10 files retrieved for both Q_1 and Q_2 contain at least one relevant file each. Therefore, TR_{10} is also 100%.

- MAP: The average precision for Q_1 and Q_2 is calculated. The sum of the average

precision is averaged over both queries to obtain the MAP as follows:

$$AP_{Q_1} = \frac{1 + 0.66 + 0.6 + 0.44}{4} = 0.68$$

$$AP_{Q_2} = \frac{0.33 + 0.29 + 0.33}{3} = 0.32$$

$$MAP = \frac{0.68 + 0.32}{2} = 0.5$$

- MRR: The rank reciprocal for each query is calculated as the reciprocal of the rank of the first relevant file. The sum of the reciprocals is then averaged over the queries to obtain MRR. Mathematically:

$$RR_{Q_1} = 1/1 = 1,\ RR_{Q_2} = 1/3 = 0.33$$

$$MRR = \frac{1 + 0.33}{2} = 0.67$$

2.3.4 Research Questions

The main objective of our experimental analysis is to evaluate the performance of the proposed co-occurrence-based methods against the baseline IR methods. Given our experimental setup, we formulate the following research questions:

- **RQ$_1$:** How effective are TSS_{PMI} and TSS_{NGD} compared to VSM, JSM, and LSI?
- **RQ$_2$:** How does the dataset size affect the performance of the different methods?
- **RQ$_3$:** How do TSS_{PMI} and TSS_{NGD} compare to each other?

2.4 Implementation, Results, and Discussion

In this section, we describe the implementation of the different proposed methods, present our analysis results, and discuss our main findings in greater detail.

```
//Sets up the attributes for the class
AddHCPAction(DAOFactory factory, long loggedInMID)
{
     this.personnelDAO = factory.getPersonnelDAO();
     this.authDAO = factory.getAuthDAO();
}
```

```
set up attribut
add hcp action dao factori factori log mid
personnel dao factori get personnel dao
auth dao factori get auth dao
```

Figure 2.2. Example: Source code indexing

2.4.1 Implementation

The first step in our analysis is to index source code. Indexing is the process of extracting and tokenizing the textual content of code embedded in identifier names, internal comments, and string literals. In our analysis, a code file is treated as a raw text file. The textual content of each file is extracted using standard string manipulation. Next, extracted code identifiers are split into constituent words using standard camel-casing (e.g., `UserID` is split to `User` and `ID`). This step can be avoided if the analyst is interested in full identifier names rather than their primitive vocabulary. Next, reserved programming words and English stop-words are filtered out (e.g., `int`, `the`). The stop-word list provided by StanfordNLP is used in our analysis. The remaining terms are stemmed to their morphological roots using Porter stemmer [97]. The outcome of the indexing process is compact content descriptors (term vectors) of each code artifact.

In our analysis, we work at a class granularity level (i.e., a code artifact is a single class). In particular, since we are dealing mainly with Object-Oriented systems, we assume that each file holds a single class. Inner classes are considered to be a part of their main class. Fig. 2.2 shows the outcome of the indexing process over a sample code segment. Specific details about the code indexing process is presented in our code indexing tool STAC [63].

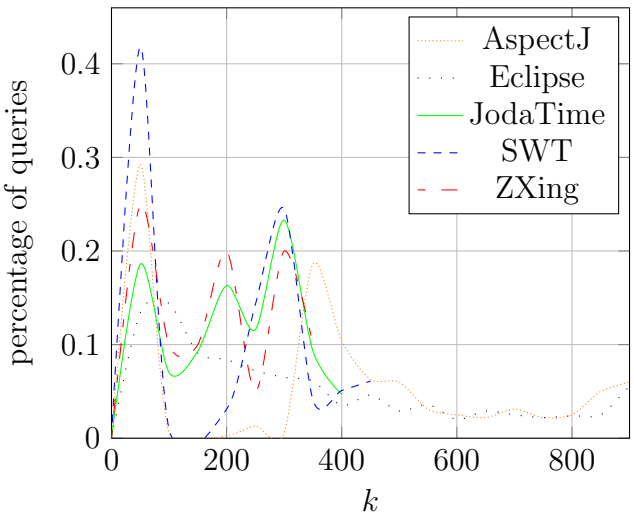
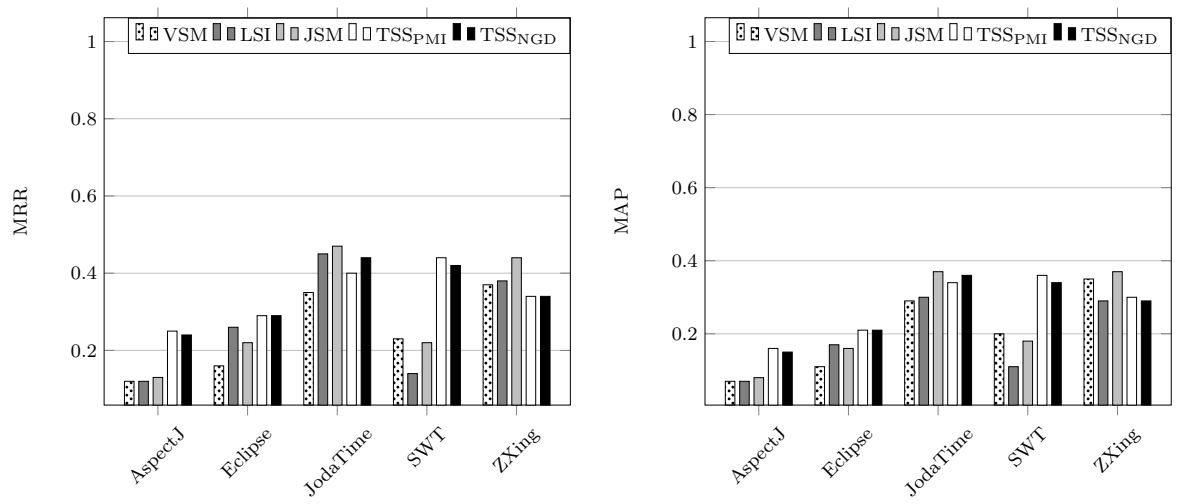


Figure 2.3. The percentage of queries for which LSI achieves the highest MRR for the corresponding value of k. Each node represents the average of the best values of MRR achieved at that k.

To implement LSI (i.e., SVD), we use The Bluebit Matrix Calculator, a high-performance matrix algebra for .NET programming. A brute-force strategy is adopted to determine LSI-k values for each of our datasets. Specifically, we run our analysis at different k values and monitor the performance. For each query in each system, we generate the LSI space using a k value in the range {50, 100, 150, 200, ..., 900}. The performance in terms of reciprocal rank is then measured for each query at each k. In Fig. 2.3, we plot the percentage of queries that achieved their highest RR values at each k value for each of our experimental datasets. The results show that the majority of queries achieved their highest RR values at k=50 in AspectJ (29.25%), SWT (41.84%), and ZXing (25.0%). In Eclipse, k=100 seems to be achieving the highest RR, as observed in 14.44% of the queries. In JodaTime, 23.26% of the queries achieved their best performance at k=300. The results also show that the performance of LSI tends to decline when $k > 400$.

To implement the co-occurrence methods, first, we extract the raw co-occurrence matrix of each system. The PMI and NGD pairwise similarity matrices are then calculated using Eq. 2.1 and Eq. 2.3 respectively. The similarity between each bug report and the source files in the corresponding system is then calculated using TSS (Eq. 2.10).



(a) MRR (b) MAP

Figure 2.4. The performance measures of the TSS methods and the baseline methods in terms of (a) MRR and (b) MAP on bug localization datasets.

2.4.2 Results

Fig. 2.4a and Fig. 2.4b show the performance of the different investigated IR methods in terms of MRR and MAP, respectively. Statistical testing is used to provide evidence that the results are unlikely to be obtained by chance. In particular, we use the Wilcoxon signed-rank test to detect statistical significance. Wilcoxon rank-sum test is a non-parametric test that makes no assumptions about the normality of the data. This test is preferred when the population (number of instances) is small that assumptions of data normality will not be necessarily true [33]. In our analysis, we apply the Wilcoxon rank-sum test to examine the significance of the differences in performance of the different investigated methods in terms of MRR and MAP. We compare the RR and AP values for each query in each dataset achieved by TSS_{PMI} and TSS_{NGD} against the RR and AP values achieved by the baseline methods (VSM, LSI, and JSM). Statistical significance is measured at $p \leq .05$. Tables 2.9 and 2.10 report the results of our statistical analysis.

In terms of MRR, the results indicate that TSS_{PMI} and TSS_{NGD} significantly outperformed the baseline methods in Eclipse, AspectJ, and SWT. In JodaTime, JSM managed to achieve the best results. However, the improvement over TSS_{PMI} and TSS_{NGD} was not statistically

Table 2.7. Comparing NGD and PMI using Wilcoxon Sum Rank Test.

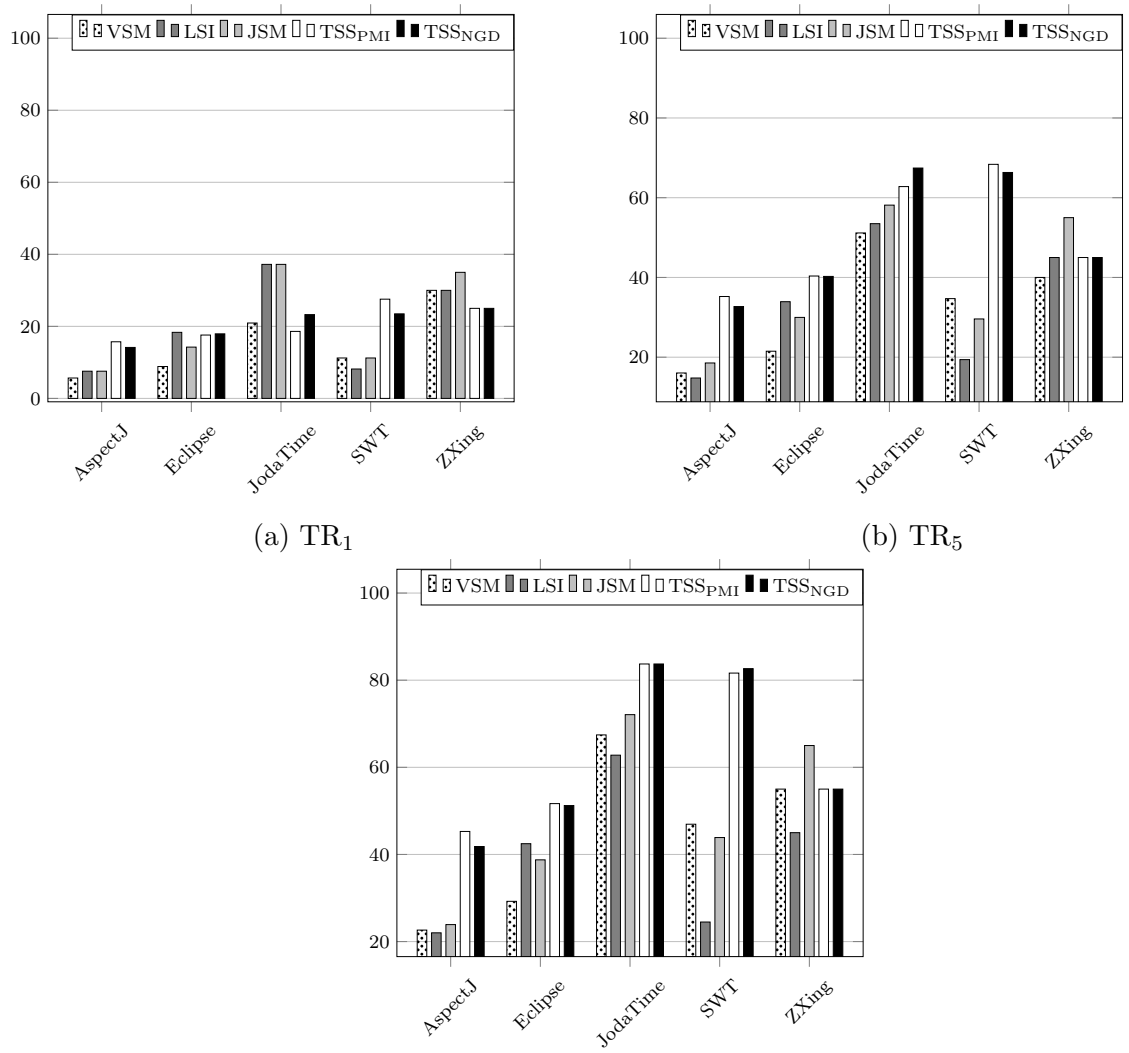
	MRR	MAP
AspectJ	**p=.000, Z=-6.569**	**p=.000, Z=-7.292**
Eclipse	p=.849, Z=-.191	p=.723, Z=-.355
JodaTime	**p=.021, Z=-2.309**	p=.158, Z=-1.411
SWT	p=.529, Z=-.630	p=.286, Z=-1.068
Zxing	p=.972, Z=-.035	p=.569, Z=-.569

significant, with TSS_{NGD} achieving a slight improvement over TSS_{PMI}. Similar behavior was observed in ZXing; the baseline methods outperformed both TSS_{PMI} and TSS_{NGD}. However, none of the performance differences were statistically significant.

In terms of MAP, the results in Fig. 2.4b and Table 2.10 show that TSS_{PMI} and TSS_{NGD} significantly outperformed the baseline methods in Eclipse, AspectJ, and SWT. In JodaTime, JSM managed to achieve the best results. However, the improvement over TSS_{PMI} and TSS_{NGD} was not statistically significant, with TSS_{NGD} achieving a slight improvement over TSS_{PMI}. In the ZXing dataset, JSM also outperformed all other methods, followed by VSM and then TSS_{PMI}, which slightly outperformed TSS_{NGD}. Again, however, none of the differences in the performance was statistically significant.

Fig. 2.5 shows the TN_1, TN_5, and TN_{10} values achieved by each investigated IR methods over all of our experimental datasets. The results show that TSS_{PMI} and TSS_{NGD} achieved the highest TN_5 and TN_{10} in all datasets except in ZXing dataset, where JSM outperformed all other methods. TN_1 has shown a different behavior. In particular, TSS_{PMI} and TSS_{NGD} outperformed the baseline methods in AspectJ and SWT. In JodaTime, LSI and JSM achieved equivalent performance, outperforming all the other methods, In ZXing, the baseline methods outperformed TSS_{PMI} and TSS_{NGD}, with JSM achieving the best results and VSM and LSI achieving equal slightly better performance, than TSS_{PMI} and TSS_{NGD}. Table 2.8 summarizes the performance of the different IR methods.

To answer $\mathbf{RQ_1}$, TSS_{PMI} and TSS_{NGD} seem to outperform the baseline methods in MAP,



(a) TR_1

(b) TR_5

(c) TR_{10}

Figure 2.5. The performance measures of the TSS methods and the baseline methods in terms of TR on bug localization datasets.

Table 2.8. The performance measures of the TSS methods and the baseline methods on bug localization datasets at 0.7 threshold.

System	Method	$\mathbf{Top_1}$	$\mathbf{Top_5}$	$\mathbf{Top_{10}}$	MRR	MAP
AspectJ	VSM	18 (05.66%)	51 (16.04%)	72 (22.64%)	00.12	00.07
	LSI	24 (07.55%)	47 (14.78%)	70 (22.01%)	00.12	00.07
	JSM	24 (07.55%)	59 (18.55%)	76 (23.90%)	00.13	00.08
	PMI	50 (15.72%)	112 (35.22%)	144 (45.28%)	00.25	00.16
	NGD	45 (14.15%)	104 (32.70%)	133 (41.82%)	00.24	00.15
Eclipse	VSM	272 (08.85%)	661 (21.50%)	899 (29.24%)	00.16	00.11
	LSI	564 (18.34%)	1043 (33.92%)	1306 (42.47%)	00.26	00.17
	JSM	438 (14.24%)	922 (29.98%)	1192 (38.76%)	00.22	00.16
	PMI	541 (17.59%)	1241 (40.36%)	1589 (51.67%)	00.29	00.21
	NGD	551 (17.92%)	1238 (40.26%)	1575 (51.22%)	00.29	00.21
JodaTime	VSM	9 (20.93%)	22 (51.16%)	29 (67.44%)	00.35	00.29
	LSI	16 (37.21%)	23 (53.49%)	27 (62.79%)	00.45	00.30
	JSM	16 (37.21%)	25 (58.14%)	31 (72.09%)	00.47	00.37
	PMI	8 (18.60%)	27 (62.79%)	36 (83.72%)	00.40	00.34
	NGD	10 (23.26%)	29 (67.44%)	36 (83.72%)	00.44	00.36
SWT	VSM	11 (11.22%)	34 (34.69%)	46 (46.94%)	00.23	00.20
	LSI	8 (08.16%)	19 (19.39%)	24 (24.49%)	00.14	00.11
	JSM	11 (11.22%)	29 (29.59%)	43 (43.88%)	00.22	00.18
	PMI	27 (27.55%)	67 (68.37%)	80 (81.63%)	00.44	00.36
	NGD	23 (23.47%)	65 (66.33%)	81 (82.65%)	00.42	00.34
ZXing	VSM	6 (30.00%)	8 (40.00%)	11 (55.00%)	00.37	00.35
	LSI	6 (30.00%)	9 (45.00%)	9 (45.00%)	00.38	00.29
	JSM	7 (35.00%)	11 (55.00%)	13 (65.00%)	00.44	00.37
	PMI	5 (25.00%)	9 (45.00%)	11 (55.00%)	00.34	00.30
	NGD	5 (25.00%)	9 (45.00%)	11 (55.00%)	00.34	00.29

Table 2.9. Comparing Methods Performance in terms of RR using Wilcoxon Sum Rank Test.

	PMI			NGD		
	VSM	LSI	JSM	VSM	LSI	JSM
AspectJ	**p=.00, Z =-9.89**	**p=.00, Z=-8.41**	**p=.00, Z=-9.08**	**p=.00, Z=-9.05**	**p=.00, Z=-7.69**	**p=.00, Z=-8.62**
Eclipse	**p=.00, Z=-26.55**	**p=.00, Z=-7.08**	**p=.00, Z=-15.28**	**p=.00, Z=-26.86**	**p=.00, Z=-7.01**	**p=.00, Z=-15.45**
JodaTime	p=.39, Z=-.87	p=.75, Z=-.32	p=.54, Z=-.611	p=.13, Z=-1.51	p=.94, Z=-.080	p=.889, Z=-.139
SWT	**p=.00, Z=-4.64**	**p=.00, Z=-6.15**	**p=.00, Z=-5.08**	**p=.00, Z=-4.32**	**p=.00, Z=-6.13**	**p=.00, Z=-5.002**
Zxing	p=.96, Z=-.05	p=.88, Z=-.15	p=.27, Z=-1.108	p=.981, Z=-.024	p=.95, Z=-.07	p=.28, Z=-1.08

Table 2.10. Comparing Methods Performance in terms of AP using Wilcoxon Sum Rank Test.

	PMI			NGD		
	VSM	LSI	JSM	VSM	LSI	JSM
AspectJ	**p=.00, Z=-10.21**	**p=.00, Z=-9.69**	**p=.00, Z=-10.12**	**p=.00, Z=-9.48**	**p=.00, Z=-8.87**	**p=.00, Z=-9.52**
Eclipse	**p=.00, Z=-29.12**	**p=.00, Z=-13.20**	**p=.00, Z=-17.73**	**p=.00, Z=-29.67**	**p=.00, Z=-13.13**	**p=.00, Z=-17.65**
JodaTime	p=.19, Z=-1.32	p=.22, Z=-1.23	p=.94, Z=-.07	p=.08, Z=-1.74	p=.10, Z=-1.63	p=.58, Z=-.55
SWT	**p=.00, Z=-4.72**	**p=.00, Z=-6.30**	**p=.00, Z=-5.26**	**p=.00, Z=-4.42**	**p=.00, Z=-6.28**	**p=.00, Z=-5.21**
Zxing	p=.78, Z=-.28	p=.74, Z=-.33	p=.11, Z=-1.60	p=.87, Z=-.17	p=.78, Z=-.28	p=.11, Z=-1.60

MRR, TR_5, and TR_{10}, achieving statistically significant improvements over the baseline methods over larger datasets. These results lead to answering $\mathbf{RQ_2}$. In particular, TSS_{PMI} and TSS_{NGD} work better in larger systems, while JSM seems to be working better for smaller datasets. Finally, in terms of the difference in performance between TSS_{PMI} and TSS_{NGD} (i.e., $\mathbf{RQ_3}$), as Table 2.7 shows, no statistically significant difference was detected between the two methods in most of the datasets. In the next section, we discuss the outcome of our analysis in greater detail.

2.4.3 Discussion

Our results show that TSS_{PMI} and TSS_{NGD} were able to outperform other baseline methods in the majority of our datasets. Even in the datasets which showed mixed results (JodaTime and ZXing), the difference in the performance failed to reach a significant level. The relatively better performance of TSS_{PMI} and TSS_{NGD} over bug localization tasks can be explained based on the observation that co-occurrence-based methods tend to favor larger files over smaller ones. In other words, larger files tend to be given a higher similarity score, thus appear higher in the retrieved list. This gives these methods a clear advantage over other IR methods in bug localization tasks as larger source code files tend to be more defect-prone [92, 135]. Zhang et al. [135] reported that a small number of the largest source files in software systems account for a large proportion of the defects. Ostrand et al. [92] made similar observations, who reported that 20% of the largest files contained around 70% of the reported bugs. These observations seem to also hold in our experimental datasets. Table 2.11 shows the percentage of defects found in each system's top n% largest files. The table shows that the largest 10% of files in all systems were responsible for an average of 64% of the defects, while the largest 20% contained an average of 84% of the bugs. Therefore, ranking larger files higher in bug localization retrieval tasks can lead to more accuracy (better bug detection) [137].

The failure of the lexical matching method VSM can be attributed to the vocabulary mismatch problem. In particular, bugs are often described using vocabulary different from

Table 2.11. Percentage of bugs consisting of at least one relevant file in the top N largest files (10%, 20%, 40%, and 80%) in the experimental systems.

System	**10%** (%)	**20%** (%)	**40%** (%)	**80%** (%)
AspectJ	83.33	94.65	99.69	100.00
Eclipse	63.71	81.01	93.89	99.71
JodaTime	51.16	72.09	81.40	97.67
SWT	77.55	91.84	98.98	100.00
ZXing	45.00	80.00	95.00	100.00

the source code. Therefore, relying solely on lexical matching (similar terms in bug reports and source code files) may lead to information loss. For example, let A and B be two source files from a certain software system, W_A is the set of words in A, and W_B is the set of words in B. Using VSM, the similarity between A and B is calculated using $W_A \cap W_B$ only. However, using a method such as PMI, the similarity between A and B is calculated using $O(W_A \cup W_B)$. In other words, even if there were no similar words between A and B (i.e., $W_A \cap W_B = Null$), a relation might be detected between the words based on their co-occurrence patterns in other artifacts in the system. Furthermore, the probability of detecting such lexically different but semantically similar terms is higher for larger files, which adds more value to the similarity scores between the two files.

Fig. 2.6 shows an example of a bug report (Bug #94459) from the Eclipse dataset. Stop-words and lemmatized characters are stroked out. A single source file, *DefaultOperationHistory.java*, was modified to fix this bug. The terms in the bug report that are also present in the relevant source file are emphasized. For this bug, VSM locates the relevant file, *DefaultOperationHistory.java*, in the 11^{th} position in its generated list of candidate relevant files and the irrelevant file *IUndoManagerExtension.java* occupies the 1^{st} rank in the list. *IUndoManagerExtension.java* consists of 82 terms and has only 4 distinct terms in common with the bug report, whereas *DefaultOperationHistory.java* consists of 3,343 terms and has 12 distinct terms in common with the bug report. The ranking of these two files can be attributed to the fact that cosine-

[**Undo**] - ~~should~~ **refactor**~~ing~~ **undo** work ~~when~~ text **undo limit** ~~is~~ ~~0~~? **Refactor**~~ing~~ **undo**~~s~~ ~~are~~ ~~not~~ **undo**~~ing~~ ~~up~~ ~~in~~ ~~the~~ ~~package~~ explore~~r~~ **undo** menu ~~when~~ ~~the~~ text **undo limit** ~~is~~ **set** ~~to~~ ~~0~~. Technic~~ally~~ ~~the~~ **refactor**~~ing~~ **operation** ~~should~~ ~~be~~ permit~~ted~~ ~~on~~ ~~the~~ workspace **undo context histor**~~y~~. ~~The~~ **histor**~~y~~ ~~should~~ ~~just~~ ~~be~~ **remov**~~ing~~ ~~the~~ text **context** ~~from~~ ~~the~~ **operation**. ~~One~~ ~~could~~ argue ~~this~~ ~~is~~ ~~a~~ **correctl**~~y~~ work~~ing~~ "feature" ~~but~~ ~~need~~ ~~to~~ ~~check~~ ~~why~~ ~~it's~~ **happen**~~ing~~ ~~and~~ resolve ~~the~~ **behavior** ~~as~~ ~~a~~ bug ~~or~~ document ~~as~~ expect~~ed~~ outcome.

Figure 2.6. An Eclipse bug report

similarity is used to compute similarity in VSM and it is inversely related to the size of the files (length of document vectors); thus, larger relevant files get assigned lower similarity than smaller, maybe irrelevant, files.

JSM attempts to overcome the vocabulary mismatch problem by considering all terms in the document to compute similarity (divergence). Therefore, larger files tend to have higher similarity scores than smaller files. Similar to JSM, TSS also computes the similarity score by using all terms in the documents. Therefore, JSM and TSS outperform VSM for most bugs. However, a drawback of JSM is that the probability distributions for terms tend to be smaller for larger files since the entropy of the distribution is computed using all terms in the document. For example, consider two terms, *refactor* and *featur*, from the above example (Fig. 2.6). The term *refactor* is an exact match (appears in the bug report and the relevant file). Using TSS, an exact match contributes the maximum score of 1 to the similarity between the two artifacts. However, using JSM, *refactor* only contributes a value of 0.00089 and 0.069 for the relevant source file and the bug report, respectively. The term *featur*, on the other hand, is only present in the bug report. Therefore, using JSM, the term only contributes to the similarity calculation from the source code file distribution. In comparison, TSS_{PMI} detected a 0.4 similarity between the term *featur* and the term *instal*, which appears in the relevant source file. This relatively high pairwise similarity between these two terms was generated because the two terms frequently appear together (co-occur) in the system. Examining the code shows that the term *install* is frequently associated with the feature *undo*, which is the main topic of the bug report.

Our results show that LSI has achieved relatively lower performance levels in all systems. This behavior can be explained based on the fact that latent methods tend to be data

extensive. In other words, larger amounts of data are required for such methods to generate meaningful semantic representations [1]. Therefore, such methods tend to fail in software systems as the data is often sparse. Furthermore, in comparison to other methods, a major drawback of LSI stems from its computationally expensive calibration process. As observed in Fig. 2.3, no single universal k value works for all systems, and often, the SVD space has to be recalculated frequently to determine k. This raises major feasibility concerns when adapting LSI to practical applications [104]. In comparison, methods that rely solely on word counts, even though the co-occurrence matrix has to be updated to accommodate new words, no additional processing over the matrix is required other than a straightforward tabulation of new co-occurrence information. Furthermore, co-concurrence-based methods do not break the bag-of-words assumption of other methods, thus keeping moderate computational and space requirements.

In summary, our experimental analysis shows that the proposed co-occurrence methods outperform other commonly-used IR methods in bug localization tasks, including VSM, JSM, and LSI. By expanding the scope of term semantic similarity calculation to the entire system, TSS_{PMI} and TSS_{NGD} can capture semantic relations between terms that other methods fail to capture. Information-theoretic co-occurrence methods provide *"just enough semantics"* necessary to establish relations between bug reports and code artifacts, achieving a balance between lexical methods that rely solely on exact matches, and more computationally-expensive semantic methods that often require substantial amounts of data to function properly.

2.5 Threats to Validity

The analysis presented in this chapter has several limitations that might affect the validity of the results. In this section, we describe our potential internal, external, construct, and conclusion validity threats along with our mitigation strategies.

2.5.1 Internal Validity

Internal validity refers to confounding factors that might affect the causal relations established in the experiment [35]. A potential threat to the proposed study's internal validity is the class-granularity level adopted in our analysis. In particular, different granularity levels might considerably change the behavior of text analysis methods. However, since we deal with Object-Oriented systems, each class supposedly encapsulates one functionality, thus can be treated as a separate document. Furthermore, the decision to consider a class level is justified by the observation that higher granularity levels (e.g., a code snippet or a method) do not provide sufficient window size to collect co-occurrence data or provide sufficient context for methods such as LSI to work [18]. On the other hand, at the package level, the window size is too big that co-occurrence cues become meaningless, thus increasing the likelihood of irrelevant and misleading information and depriving PMI, NGD, and LSI of context. Nonetheless, we believe that a future study dedicated to investigating the effect of granularity level, or even other parameters such as the size of the system, on the performance of the proposed approach is necessary to confirm our observations further.

An internal validity argument could be made about using a brute-force strategy to calibrate a method such as LSI. In particular, other calibration strategies which use Genetic algorithms or Neural Networks might provide a more efficient solution to the problem [75]. Such strategies become especially useful when dealing with many features that make a brute-force solution unfeasible. However, we only consider one calibration parameter (k) in our analysis, making an exhaustive search strategy feasible. Furthermore, such a strategy is guaranteed to find a solution if a solution exists in the k space.

2.5.2 External Validity

Threats to external validity impact the generalizability of results. In particular, the results of our experiment might not generalize beyond the specific experimental settings used in this paper [35]. A potential threat to our external validity stems from the datasets used in our experiment. In particular, our datasets are limited in size. To mitigate this threat, we

compiled our ground-truth dataset from several sources, including five benchmark datasets derived from different size systems that have been used before in the literature. Using such publicly available datasets enables other researchers to replicate our results independently.

Furthermore, our results might not generalize beyond our specific implementations of the different methods investigated in this chapter. While this threat is inevitable, we hope that other researchers can replicate our analysis and compare our implementation to theirs' by making our implementations public.

2.5.3 Construct Validity

Construct validity is the degree to which the various performance variables accurately measure the concepts they purport to measure [35]. In our experiment, there were minimal threats to construct validity as the standard performance measures, extensively used in related research, were used to assess the performance of different methods. We believe that these measures sufficiently captured and quantified the different aspects of performance.

2.5.4 Conclusion Validity

Threats to the conclusion validity are concerned with issues that affect the ability to draw the correct conclusion about relations between the treatment and the outcome of an experiment [125]. The main conclusion threat might stem from the low statistical power of the test used to reveal the patterns in the data. In our analysis, we used Wilcoxon signed-rank test. This test is commonly used in bug localization and traceability research to conclude [50, 65, 80, 121, 122]. In these tasks, this test is preferred over parametric tests such as ANOVA as it does not make any assumptions about the normality of the data. Thus no assumptions of the statistical test are violated.

2.6 Conclusion

Bug reports, often expressed in free language and code fragments, contain information about the bug in the source code. Such information can be exploited by IR methods to establish an association between a bug report and the source code fragments responsible for the bug. Following this line of research, we investigated the performance of a new paradigm

of information-theoretic IR methods in bug localization tasks. The investigated methods, including Pointwise Mutual Information (PMI) and Normalized Google Distance (NGD), rely on the contextual cues of terms in software systems to establish semantic relatedness between code terms. Such information was then used to establish association relations between source code artifacts and bug reports using the textual semantic similarity (TSS). The main assumption is that co-occurrence methods can capture semantic dimensions that other methods often fail to detect.

Five benchmark datasets from different application domains were used to conduct our analysis. The proposed methods, including TSS_{NGD} and TSS_{PMI}, were compared against multiple IR methods that are commonly used in bug localization research, including Vector Space Model (VSM), Jensen-Shannon Model (JSM), and Latent Semantic Indexing (LSI). The results showed that TSS was more successful than other methods in locating buggy source files in larger datasets, achieving statistically significant improvements over other baseline methods. In particular, term co-occurrence-based methods were more capable of overcoming the vocabulary mismatch problem affecting methods such as VSM. At the same time, co-occurrence-based methods avoid the need for exhaustive calibration that is often required for semantically aware methods, such as LSI, to achieve adequate accuracy.

Chapter 3
Combining IR Methods to Improve BL

Information Retrieval (IR) methods have been recently employed to provide automatic support for bug localization tasks. However, for an IR-based bug localization tool to be useful, it must achieve adequate retrieval accuracy. Lower precision and recall can leave developers with large amounts of incorrect information to wade through. To address this issue, in this chapter, we systematically investigate the impact of combining various IR methods on the retrieval accuracy of bug localization engines. The main assumption is that different IR methods, targeting different dimensions of similarity between artifacts, can enhance the confidence in each others' results. Five benchmark systems from different application domains are used to conduct our analysis. The results show that *a)* near-optimal global configurations can be determined for different combinations of IR methods, *b)* optimized IR-hybrids can significantly outperform individual methods and other unoptimized methods, and *c)* hybrid methods achieve their best performance when utilizing information-theoretic IR methods. Our findings can be used to enhance the practicality of IR-based bug localization tools and minimize the cognitive overload developers often face when locating bugs.

3.1 Introduction

IR-based bug localization methods are appealing for their low computational cost and the fact that they can be independent of the programming language [12, 76, 84, 104, 137, 70]. However, due to their inherent sub-optimal accuracy, IR methods are still far from achieving performance levels that are adequate for practical applications. Consequently, developers still have to vet the outcomes of these tools to locate relevant code (true positives) and discard incorrect matches (false positives).

To improve the performance of existing IR-based bug localization tools, researchers have considered combining various IR methods into hybrid pairs [13, 47, 50, 115, 121]. The hybrid approach can be analogous to consulting multiple experts with different types of expertise. In particular, combining orthogonal IR methods, which target different aspects

of similarity between text artifacts, is expected to enhance the confidence in the retrieved results (precision) and help retrieve links missed by individual methods (recall). However, existing hybrid methods treat individual IR methods equally or rely on existing heuristics to assign confidence levels to each method in the hybrid combination [50, 115]. These tactics can limit the improvement in the performance as different IR methods perform differently; thus, they should not be equally trusted.

To overcome these limitations, in this paper, we propose an optimized approach for systematically combining individual IR methods into hybrid pairs. Our approach is calibrated based on the performance of individual IR methods. To conduct our analysis, we experiment with four different IR methods that employ different mechanisms for calculating the textual similarity between textual artifacts [10, 21, 50, 64, 76, 90, 104, 115, 130, 137]. Specifically, our set of IR methods consists of Vector Space Model (VSM) [105] as a representative of string matching methods, Latent Semantic Indexing (LSI) [36] as a representative of semantically-enabled methods, Jensen-Shannon Model (JSM) [1] as a representative of probabilistic methods, and Pointwise Mutual Information (PMI) [23] as a representative of information-theoretic text retrieval methods.

Our analysis is conducted using five benchmark systems obtained from a broad range of application domains. Our objectives are to ***a)*** determine the nature of the impact that combining individual IR methods has on the retrieval accuracy during bug localization tasks, ***b)*** approximate configuration settings that can be used to optimize the performance of the hybrid methods, ***c)*** compare the performance of optimized versus unoptimized IR-hybrid methods, and ***d)*** study the impact of individual methods' performance on the performance of their combinations. In what follows, we introduce the hybrid IR methods, describe our experimental setup, present our results, and discuss our main findings.

3.2 The Hybrid Approach

The basic idea behind the hybrid approach is that different IR methods use different expertise to localize bugs. In other words, each method infers the similarity judgment between

a bug report and code artifacts differently. For instance, VSM uses simple word matching to calculate similarity, while LSI calculates the similarity between texts by comparing their underlying latent semantic structures (or topics). Furthermore, the hybrid approach assumes that each IR method is an independent expert on bug localization whose judgment determines the occurrence of a bug in a code artifact. Therefore, combining experts' forecasts is expected to provide a more certain forecast than each expert [61, 124]. This approach has been successfully employed in feature location [98] and traceability link recovery [50]. Our analysis considers two types of hybrid approaches: optimized and unoptimized.

3.2.1 Optimized IR-Hybrids

A hybrid approach of IR methods can be obtained by combining the ranked lists generated by different IR methods based on some sort of an optimization function [50]. Formally, let X be the set of source code files in the system, and let Y be the list of bug reports. Let i and j be any two IR methods, $sim_i(x, y)$ is the similarity between the source code artifact x and the bug report y as calculated by the IR method i, and λ is an optimization variable which expresses the confidence in the judgment of the method i. The combination of i and j can be established as follows:

$$sim_{i,j}(x, y) = \lambda \times sim_i(x, y) + (1 - \lambda) \times sim_j(x, y) \tag{3.1}$$

Eq. 3.1 assumes that the similarity scores generated by methods i and j are in same range (e.g., [0-1]). If this is not the case, the scores can be normalized as follows:

$$sim_i(x, y)_{normalized} = \frac{sim_i(x, y) - \overline{sim_i(X, Y)}}{\sigma(sim_i(X, Y))} \tag{3.2}$$

where $\overline{sim_i(X, Y)}$ is the average similarity scores of all source code artifacts and bug reports in the system and $\sigma(sim_i(X, Y))$ is the standard deviation of the similarity scores.

Example: To demonstrate the operation of the calibrated hybrid approach, consider the retrieval scenario in Table 3.1. This table shows the ranked list of source code files generated

Table 3.1. Example: Illustration of the hybrid combination of two IR methods.

PMI			VSM			PMI-VSM Hybrid			Rank Change	
Rank	**File**	**Similarity**	**Rank**	**File**	**Similarity**	**Rank**	**File**	**Similarity**	**PMI**	**VSM**
1	Main	0.652	1	AfterWeaveTestCase	0.312	1	BuildProgressPanel	0.551	↑	←
2	TreeViewBuildConfigEditor	0.652	2	BeforeWeaveTestCase	0.299	2	TreeViewBuildConfigEditor	0.547	–	←
3	BuildProgressPanel	0.645	3	BuildManager	0.286	3	BuildNotifierAdapter	0.546	↑	↑
4	OptionsFrame	0.644	4	IClassWeaver	0.258	4	UpdateConfigurationDialog	0.533	↑	←
5	UpdateConfigurationDialog	0.640	5	BuildSpec	0.257	5	Main	0.526	↓	←
6	BuildNotifierAdapter	0.630	6	TestBuildListener	0.250	6	CompilerAdapter	0.526	↑	←
7	CompilerAdapter	0.630	7	AroundArgsWeaveTestCase	0.249	7	OptionsFrame	0.525	↓	←
8	AjBuildManager	0.623	8	BuildModule	0.244	8	AjBuildManager	0.524	–	←
9	AjcTask	0.618	9	BuildListener	0.220	9	BuildSpec	0.523	←	↓
10	BuildConfigurationTests	0.612	10	BuildNotifierAdapter	0.211	10	BuildManager	0.512	←	↓

for bug report `#40192` from the *AspectJ* system by PMI and VSM. For each retrieved file in the ranked list, the table shows the rank of the file and the similarity score between the file and the bug report as calculated by the IR method. In this example, PMI captured the two relevant (true positive) files, *BuildNotifierAdapter* at rank 6 and *AjBuildManager* at rank 8. VSM, however, was able to capture only one relevant file, *BuildNotifierAdapter*, at rank 10. These files are highlighted in the table.

The ranked lists generated by the two IR methods can be combined into a hybrid ranked list using Eq. 3.1. Assuming a λ value of 0.8 is assigned to PMI (PMI is given a higher confidence level than VSM), the similarity score for each relevant file in both ranked lists can be calculated as follows:

- *BuildNotifierAdapter.java*

  $= \lambda \times sim_{PMI} + (1 - \lambda) \times sim_{VSM}$

  $= 0.8 \times 0.630 + (1 - 0.8) \times 0.211$

  $= 0.546$

- *AjBuildManager.java*

  $= \lambda \times sim_{PMI} + (1 - \lambda) \times sim_{VSM}$

  $= 0.8 \times 0.623 + (1 - 0.8) \times 0.126$

  $= 0.524$

The hybrid ranked list generated from combining the ranked lists of PMI and VSM is

shown in the third part of Table 3.1. The last two columns of the table show the change in the files' ranking compared to their actual ranking by PMI and VSM. Compared to the original PMI's ranked list, the relevant file *BuildNotificationAdapter* climbed from the sixth position to the third in the new ranked list. On the other hand, the ranking of the file *AjBuildManager* remained the same. These results are considered an enhancement when compared to the original ranked list. Specifically, IR methods that place relevant artifacts at higher ranks are considered superior [93]. In the case of VSM, the ranking of the relevant files, *BuildNotificationAdapter* and *AjBuildManager*, have enhanced, moving from the tenth position to the third and from the thirty-third position to the eighth, respectively, thus enhancing the overall results compared to VSM alone.

3.2.2 Unoptimized Hybrid Methods

This set of hybrid methods does not apply any optimization to combine the outcome of different IR methods (i.e., individual IR methods are treated equally). In our analysis, we consider two unoptimized methods that were proven to enhance the performance of bug localization tools [115]. These methods can be described as follows:

- **Borda Count**: Borda Count [43] is a rank-only combination approach that assigns scores to the retrieved links based on their ranks in each IR method's ranked list. Formally, assuming a set of IR methods C. Each method $c_i \in C$ ranks the link k at rank $r_{i,k}$. Let M_i be the number of links that received a non-zero score by c_i. Then, the Borda Count for k in c_i is calculated as $M_i - r_{i,k}$. The total Borda Count for k in the combination of the methods in C can be calculated as:

$$Borda(k) = \sum_{i=0}^{|C|} M_i - r_{i,k} \tag{3.3}$$

  After calculating the Borda scores for all retrieved links, the rank of each link in the combined list is calculated based on its total Borda Count. For example, in Table 3.1, *BuildNotifierAdapter* (entity k) is ranked 6^{th} in the list generated by PMI and 10^{th} in

the list generated by VSM. Therefore, this file has a Borda Count of 4 (10 - 6) and 0 (10 - 10) in PMI and VSM, respectively, and a combined Borda Count of 4 (4 + 0).

- **ScoreAddition**: ScoreAddition is a score-based combination approach that sums up the scores assigned by each IR method to each retrieved link. Assuming a set of IR methods C, where each methods $c_i \in C$ assigned a score of $s_{i,k}$ to the link k. Then the Score Addition of k for the combination of IR methods in C is computed as:

$$ScoreAddition(k) = \sum_{i=0}^{|C|} s_{i,k} \tag{3.4}$$

  For example, in the lists in Table 3.1, the ScoreAddition of the file *BuildNotifierAdapter* is the summation of its score in PMI and VSM, 0.630 + 0.211 = 0.841.

3.3 Experimental Setup

In this section, we describe our main research questions. The experimental dataset, baseline IR methods, and the evaluation metrics used to assess the performance of the individual methods and the combined methods are described in Chapter 2.

3.3.1 Research Questions

The main objective of our empirical investigation is to compare the performance of the λ-optimized IR-hybrids against the individual IR methods as well as the unoptimized IR-hybrids (Borda Count and Score Addition). To guide our analysis, we formulate the following research questions:

- **RQ_1:** Is there any global optimal λ that can be used for combining IR methods?
- **RQ_2:** How effective is the hybrid approach compared to the individual IR methods?
- **RQ_3:** How effective are the λ-optimized hybrids compared to the unoptimized hybrids?
- **RQ_4:** How does the performance of individual IR methods affect the performance of their hybrid pairs?

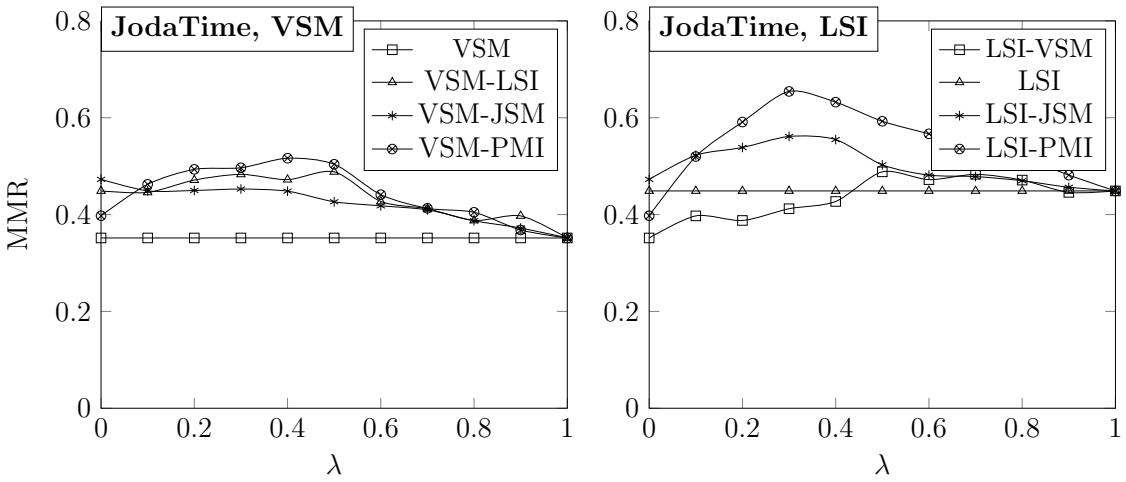


Figure 3.1. Examples of optimizing λ for VSM and LSI.

3.4 Implementation, Results, and Discussion

In this section, we describe the optimization process, present the results of the hybrid approach over our experimental systems, and discuss the results. The process of indexing the source code and implementing individual IR methods is described in Chapter 2.

3.4.1 Optimized Hybrids *vs.* Individual Methods

When combining two IR methods using the optimized approach, the main question is to determine the λ value that can maximize the retrieval accuracy of the hybrid pair. In the literature, such a value is often chosen to be 0.5. In other words, the same confidence level is assigned for both methods, i and j, in Eq. 3.1 [50, 61]. However, our analysis of individual methods' performances revealed that different methods performed differently; thus, they cannot be treated equally; instead, they should be assigned confidence levels according to their performance. To determine such values, we followed an exhaustive search approach. In particular, for each combination of methods, we measured the performance in terms of MRR at different λ values with a 0.1 step size (i.e., $\lambda = 0, 0.1, 0.2, 0.3, ..., 1$). Values of λ that maximized the performance of each hybrid pair of methods over each system were then averaged. For example, Fig. 3.1 shows the performance (MRR) of VSM and LSI when combined with other methods at different λ values over the system *JodaTime*. The solid horizontal line in the chart shows the performance of VSM before being combined with any

Table 3.2. The performance gain (%) of the hybrid methods compared to the individual IR methods.

System	Method	%change in MRR				%change in MAP				%change in TR_{10}			
		VSM	**LSI**	**JSM**	**PMI**	**VSM**	**LSI**	**JSM**	**PMI**	**VSM**	**LSI**	**JSM**	**PMI**
VSM	AspectJ	-	20.17	27.04	104.80	-	14.42	23.60	110.92	-	5.56	18.06	84.72
	Eclipse	-	58.35	35.32	85.74	-	56.90	39.36	101.38	-	44.44	28.44	76.78
	Joda	-	38.83	28.72	40.27	-	12.94	22.55	36.99	-	0.00	10.34	20.69
	SWT	-	-3.47	17.32	114.53	-	-11.53	14.90	103.66	-	-21.74	17.39	76.09
	ZXing	-	28.56	19.88	-1.70	-	9.31	18.11	-10.83	-	-9.09	18.18	9.09
Average		-	28.49	25.66	68.73	-	16.41	23.70	68.42	-	3.83	18.48	53.47
LSI	AspectJ	13.46	-	16.67	115.99	19.77	-	23.48	148.46	8.57	-	20.00	98.57
	Eclipse	-6.42	-	3.29	38.09	-1.69	-	10.26	58.13	-0.46	-	6.74	41.88
	Joda	8.79	-	25.06	31.70	8.93	-	30.96	47.00	7.41	-	18.52	37.04
	SWT	56.79	-	43.66	212.39	64.40	-	48.78	243.43	50.00	-	54.17	250.00
	ZXing	24.21	-	19.53	26.07	31.14	-	24.68	36.41	11.11	-	33.33	44.44
Average		19.36	-	21.64	84.85	24.51	-	27.63	106.69	15.33	-	26.55	94.39
JSM	AspectJ	13.75	10.69	-	74.55	14.07	8.94	-	78.23	11.84	10.53	-	57.89
	Eclipse	-5.58	21.96	-	37.24	-5.82	18.94	-	43.95	-3.02	16.95	-	38.26
	Joda	-4.23	18.74	-	28.18	-3.15	7.30	-	27.96	3.23	3.23	-	25.81
	SWT	27.96	-3.53	-	109.01	26.47	-12.01	-	106.47	25.58	-13.95	-	90.70
	ZXing	1.08	4.31	-	-6.11	10.47	-2.80	-	-7.31	0.00	-7.69	-	0.00
Average		6.60	10.43	-	48.58	8.41	4.08	-	49.86	7.53	1.81	-	42.53
PMI	AspectJ	-6.75	4.91	-11.24	-	-5.91	7.26	-13.84	-	-7.64	-3.47	-16.67	-
	Eclipse	0.69	26.71	6.68	-	0.25	25.69	6.10	-	0.13	16.68	3.78	-
	Joda	24.09	48.69	52.42	-	17.30	30.50	38.64	-	-2.78	2.78	8.33	-
	SWT	15.28	3.35	2.97	-	13.25	2.77	4.51	-	1.25	5.00	2.50	-
	ZXing	5.96	40.66	20.04	-	5.88	35.01	17.68	-	9.09	18.18	18.18	-
Average		7.85	24.86	14.18	-	6.15	20.25	10.62	-	0.01	7.83	3.22	-

Table 3.3. Average near-optimal value of λ that results in the best performance (MRR) of the hybrid methods across all experimental systems.

	VSM	**LSI**	**JSM**	**PMI**
VSM	-	0.5	0.3	0.2
LSI	0.5	-	0.3	0.2
JSM	0.7	0.7	-	0.5
PMI	0.8	0.8	0.5	-

other method. Other lines show the performance of VSM, in terms of MRR, when combined with other methods using different values of λ.

The optimal average values of λ for each method pair are shown in Table 3.3. The table shows that, when combining VSM and LSI, both methods can be assigned an equal λ value. In other words, both methods can be trusted equally. However, if VSM is to be combined with PMI, then VSM should be assigned lower confidence ($\lambda = 0.2$) while PMI is assigned higher confidence (1 - $\lambda = 0.8$). In general, our results show that PMI is to be trusted the most, followed by JSM, LSI, and finally VSM. These results are aligned with our individual methods' analysis results (PMI achieved the best performance individually, followed by JSM, LSI, and finally VSM). These results answer our first research question ($\mathbf{RQ_1}$).There is no global optimal λ value that works for all combinations of the IR methods. The value of λ should vary depending on the performance of the methods to be combined.

To answer our second research question ($\mathbf{RQ_2}$), we generate the ranked lists for each system using all combinations of the IR methods given the λ values identified earlier. The results are shown in Table 3.2. The table shows the percentage change (potential loss or gain) in MRR. Specifically, assuming the method i achieves a value of P_i for a certain performance measure, and the value $P_{i,j}$ for the same measure after being combined with method j, then the percentage change in the performance is calculated as follows:

$$\frac{P_{ij} - P_i}{P_i} \times 100\% \tag{3.5}$$

For example, in Table 3.2, the first row shows the percentage change in the values of MRR for VSM when combined with LSI, JSM, and PMI over the *AspectJ* project (VSM is the method i in Eq. 3.1). The row in the table shows that the MRR for the VSM-LSI pair increased by 20.17% compared to the MRR achieved by VSM alone. TWe used Wilcoxon signed-rank test to detect statistical significance in the loss/gain results. The results in Table 3.8, 3.9, and 3.10 show that, except for a few cases, all methods experienced significant improvement in their MRR when combined with other methods using the optimized approach. Similarly, most optimized hybrid methods experienced significant improvement in their TR_{10} scores. These results answer $\mathbf{RQ_2}$.

3.4.2 Optimized vs. Unoptimized Methods

To answer our third research question ($\mathbf{RQ_3}$), we compare the performance of the λ-optimized methods to Score Addition and Borda Count. Performance is measured using TR_1, TR_5, and TR_{10}. These measures are more sensitive for detecting critical improvements in terms of ranking true positives between the different methods. The results of our analysis are shown in Table 3.4. On average, the optimized approach outperformed the other unoptimized methods that treat all individual IR methods equally. We used McNemar's test [41] to test for statistical significance. This test uses a categorical explanatory variable to show if paired observations in two groups differ significantly in the dependent variable. Table 3.5, Table 3.6, and Table 3.7 show that the improvement of the λ-optimized method over Addition Score and Broda Count is significant in most cases, particularly for the PMI-VSM and PMI-LSI pairs. This observation can be attributed to the fact that the proposed λ-optimized method assigned more confidence to PMI, a high-performing method, over VSM or LSI, comparatively low performing methods. Additionally, it can be observed that the performance improvement is significant for *Eclipse*,. where there is a larger number of bugs to experiment with compared to the other four systems used in our experiment.

Table 3.4. TR_1, TR_5, and TR_{10} for ScoreAddition, Borda Count, and the λ-optimized approach for all combinations of IR methods used in our experiment.

Method	System	ScoreAddition			Borda Count			λ-optimized		
		TR_1	TR_5	TR_{10}	TR_1	TR_5	TR_{10}	TR_1	TR_5	TR_{10}
PMI-VSM	SWT	19.39	45.92	71.43	26.53	61.22	78.57	30.61	68.37	84.69
	ZXing	30.00	60.00	65.00	20.00	55.00	60.00	25.00	60.00	60.00
	Joda	25.58	55.81	74.42	30.23	67.44	79.07	37.21	62.79	79.07
	AspectJ	8.81	21.38	29.25	12.26	27.67	38.05	12.58	28.93	38.05
	Eclipse	11.15	26.96	36.29	13.85	31.90	44.85	15.54	37.14	48.33
Average		18.99	42.02	55.28	20.58	48.65	60.11	24.19	51.45	62.03
PMI-LSI	SWT	.00	16.33	44.90	2.04	14.29	24.49	23.47	57.14	80.61
	ZXing	40.00	50.00	65.00	30.00	55.00	60.00	35.00	60.00	65.00
	Joda	51.16	69.77	79.07	46.51	72.09	86.05	44.19	79.07	86.05
	AspectJ	11.01	23.58	33.96	15.09	27.36	37.42	17.30	34.59	43.71
	Eclipse	22.63	44.16	52.62	24.23	45.04	56.65	24.65	50.21	60.29
Average		24.96	40.77	55.11	23.57	42.76	52.92	28.92	56.20	67.13
PMI-JSM	SWT	20.41	57.14	71.43	23.47	55.10	72.45	20.41	57.14	71.43
	ZXing	30.00	65.00	65.00	25.00	60.00	60.00	30.00	65.00	65.00
	Joda	44.19	76.74	93.02	37.21	79.07	86.05	44.19	76.74	93.02
	AspectJ	12.58	25.79	33.33	12.89	25.16	34.91	12.58	25.79	33.33
	Eclipse	18.89	38.41	49.66	17.27	36.29	47.64	18.89	38.41	49.66
Average		25.21	52.62	62.49	23.17	51.12	60.21	25.21	52.62	62.49
VSM-LSI	SWT	7.14	19.39	25.51	1.02	4.08	9.18	7.14	19.39	25.51
	ZXing	40.00	50.00	55.00	40.00	50.00	55.00	40.00	50.00	55.00
	Joda	27.91	58.14	67.44	34.88	55.81	69.77	27.91	58.14	67.44
	AspectJ	7.86	17.30	24.53	6.92	16.04	23.58	7.86	17.30	24.53
	Eclipse	13.79	30.70	39.28	15.02	30.60	40.91	13.79	30.70	39.28
Average		19.34	35.10	42.35	19.57	31.31	39.69	19.34	35.10	42.35
VSM-JSM	SWT	16.33	36.73	53.06	13.27	31.63	53.06	15.31	35.71	55.10
	ZXing	35.00	50.00	65.00	30.00	50.00	65.00	35.00	50.00	65.00
	Joda	30.23	55.81	72.09	23.26	60.47	69.77	32.56	62.79	74.42
	AspectJ	8.81	17.92	26.10	7.86	18.24	26.42	8.81	18.55	26.73
	Eclipse	11.71	26.02	34.99	11.28	25.59	35.84	13.01	28.75	37.59
Average		20.41	37.30	50.25	17.13	37.19	50.02	20.94	39.16	51.77
LSI-JSM	SWT	2.04	11.22	23.47	1.02	7.14	11.22	3.06	22.45	32.65
	ZXing	40.00	45.00	50.00	35.00	50.00	65.00	40.00	50.00	65.00
	Joda	48.84	65.12	69.77	32.56	65.12	74.42	41.86	67.44	76.74
	AspectJ	8.81	18.55	26.42	10.06	17.92	24.21	8.81	19.18	25.47
	Eclipse	17.92	37.79	45.59	17.95	35.48	45.24	17.46	36.36	45.27
Average		23.52	35.54	43.05	19.32	35.13	44.02	22.24	39.09	49.03

Table 3.5. Comparing the performance of the ScoreAddition and Borda Count methods with the λ-Optimized method in terms of TR_1 using McNemar's Test (bold indicates significant improvement).

System	Addition						Borda Count					
	PMI			VSM		LSI	PMI			VSM		LSI
	VSM	LSI	JSM	LSI	JSM	JSM	VSM	LSI	JSM	LSI	JSM	JSM
SWT	$p < 0.05$	$p < 0.05$	1.00	1.00	1.00	1.00	0.45	$p < 0.05$	0.55	0.07	0.63	0.50
ZXing	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00
JodaTime	0.13	0.45	1.00	1.00	1.00	0.38	0.45	1.00	0.45	0.38	0.13	0.29
AspectJ	$p < 0.05$	$p < 0.05$	1.00	1.00	1.00	1.00	1.00	0.19	1.00	0.58	0.45	0.39
Eclipse	$p < 0.05$	$p < 0.05$	1.00	1.00	$p < 0.05$	0.24	$p < 0.05$	0.48	$p < 0.05$	$p < 0.05$	$p < 0.05$	0.25

Table 3.6. Comparing the performance of the ScoreAddition and Borda Count methods with the λ-Optimized method in terms of TR_5 using McNemar's Test (bold indicates significant improvement).

System	Addition						Borda Count					
	PMI			VSM		LSI	PMI			VSM		LSI
	VSM	LSI	JSM	LSI	JSM	JSM	VSM	LSI	JSM	LSI	JSM	JSM
SWT	$p < 0.05$	$p < 0.05$	1.00	1.00	1.00	$p < 0.05$	0.07	$p < 0.05$	0.73	$p < 0.05$	0.22	$p < 0.05$
ZXing	1.00	0.63	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00
JodaTime	0.25	0.22	1.00	1.00	0.25	1.00	0.63	0.38	1.00	1.00	1.00	1.00
AspectJ	$p < 0.05$	$p < 0.05$	1.00	1.00	0.63	0.77	0.52	$p < 0.05$	0.79	0.34	1.00	0.39
Eclipse	$p < 0.05$	$p < 0.05$	1.00	1.00	$p < 0.05$	$p < 0.05$	$p < 0.05$	$p < 0.05$	$p < 0.05$	0.87	$p < 0.05$	$p < 0.05$

Table 3.7. Comparing the performance of the ScoreAddition and Borda Count methods with the λ-Optimized method in terms of TR_{10} using McNemar's Test (bold indicates significant improvement).

System	Addition						Borda Count					
	PMI			VSM		LSI	PMI			VSM		LSI
	VSM	LSI	JSM	LSI	JSM	JSM	VSM	LSI	JSM	LSI	JSM	JSM
SWT	$p < 0.05$	$p < 0.05$	1.00	1.00	0.63	$p < 0.05$	0.07	$p < 0.05$	1.00	$p < 0.05$	0.63	$p < 0.05$
ZXing	1.00	1.00	1.00	1.00	1.00	0.25	1.00	1.00	1.00	1.00	1.00	1.00
JodaTime	0.63	0.38	1.00	1.00	1.00	0.25	1.00	1.00	0.25	1.00	0.64	1.00
AspectJ	$p < 0.05$	$p < 0.05$	1.00	1.00	0.73	0.45	1.00	$p < 0.05$	0.30	0.55	1.00	0.29
Eclipse	$p < 0.05$	$p < 0.05$	1.00	1.00	$p < 0.05$	0.45	$p < 0.05$	$p < 0.05$	$p < 0.05$	$p < 0.05$	$p < 0.05$	1.00

Table 3.8. Comparing performance of the hybrid and individual methods in terms of RR using Wilcoxon Signed Rank Test.

System	VSM			LSI			JSM			PMI		
	LSI	JSM	PMI	VSM	JSM	PMI	VSM	LSI	PMI	VSM	LSI	JSM
AspectJ	p = 0.89 Z = -0.14 ↑	p = 0.08 Z = -1.74 ↑	p < 0.01 Z = -12.20 ↑	p < 0.01 Z = -6.63 ↑	p < 0.01 Z = -5.06 ↑	p < 0.01 Z = -12.38 ↑	p < 0.01 Z = -5.47 ↑	p < 0.05 Z = -2.30 ↑	p < 0.01 Z = -12.46 ↑	p < 0.01 Z = -2.93 ↓	p = 0.29 Z = -1.05 ↑	p < 0.01 Z = -4.16 ↓
Eclipse	p < 0.01 Z = -31.14 ↑	p < 0.01 Z = -23.15 ↑	p < 0.01 Z = -34.61 ↑	p = 0.74 Z = -0.33 ↓	p < 0.01 Z = -8.73 ↑	p < 0.01 Z = -23.79 ↑	p = 0.83 Z = -0.21 ↓	p < 0.01 Z = -21.50 ↑	p < 0.01 Z = -28.10 ↑	p = 0.84 Z = -0.20 ↑	p < 0.01 Z = -19.31 ↑	p < 0.01 Z = -4.41 ↑
JodaTime	p = 0.09 Z = -1.70 ↑	p < 0.01 Z = -3.55 ↑	p < 0.05 Z = -2.20 ↑	p = 0.13 Z = -1.53 ↑	p < 0.01 Z = -2.90 ↑	p < 0.01 Z = -2.62 ↑	p = 0.95 Z = -0.06 ↓	p = 0.21 Z = -1.26 ↑	p < 0.01 Z = -2.78 ↑	p < 0.05 Z = -2.22 ↑	p < 0.01 Z = -3.45 ↑	p < 0.01 Z = -3.53 ↑
SWT	p < 0.01 Z = -2.71 ↓	p < 0.05 Z = -2.16 ↑	p < 0.01 Z = -6.62 ↑	p < 0.01 Z = -7.17 ↑	p < 0.01 Z = -6.60 ↑	p < 0.01 Z = -7.55 ↑	p < 0.01 Z = -4.47 ↑	p < 0.05 Z = -2.09 ↓	p < 0.01 Z = -7.13 ↑	p < 0.05 Z = -2.09 ↑	p = 0.91 Z = -0.12 ↑	p = 0.80 Z = -0.26 ↑
ZXing	p = 0.59 Z = -0.53 ↑	p < 0.05 Z = -1.98 ↑	p = 0.83 Z = -0.21 ↓	p < 0.01 Z = -2.97 ↑	p < 0.01 Z = -2.76 ↑	p = 0.15 Z = -1.45 ↑	p = 0.37 Z = -0.90 ↑	p = 0.39 Z = -0.86 ↑	p = 0.69 Z = -0.40 ↓	p < 0.05 Z = -2.13 ↑	p < 0.05 Z = -2.24 ↑	p < 0.05 Z = -2.14 ↑

Table 3.9. Comparing the performance of the hybrid and individual methods in terms of MAP using Wilcoxon Signed Rank Test.

System	VSM			LSI			JSM			PMI		
	LSI	JSM	PMI	VSM	JSM	PMI	VSM	LSI	PMI	VSM	LSI	JSM
AspectJ	p < 0.05 Z = -2.34 ↑	p = 0.19 Z = -1.32 ↑	p < 0.01 Z = -12.66 ↑	p < 0.01 Z = -9.52 ↑	p < 0.01 Z = -7.94 ↑	p < 0.01 Z = -13.83 ↑	p < 0.01 Z = -6.31 ↑	p = 0.22 Z = -1.23 ↑	p < 0.01 Z = -13.35 ↑	p < 0.01 Z = -2.98 ↓	p = 0.54 Z = -0.61 ↑	p < 0.01 Z = -4.54 ↓
Eclipse	p < 0.01 Z = -27.70 ↑	p < 0.01 Z = -24.68 ↑	p < 0.01 Z = -37.56 ↑	p < 0.01 Z = -7.39 ↓	p < 0.01 Z = -16.90 ↑	p < 0.01 Z = -30.53 ↑	p = 0.92 Z = -0.10 ↓	p < 0.01 Z = -16.19 ↑	p < 0.01 Z = -31.32 ↑	p = 0.38 Z = -0.89 ↑	p < 0.01 Z = -18.94 ↑	p < 0.01 Z = -5.31 ↑
JodaTime	p = 0.99 Z = -0.01 ↑	p < 0.01 Z = -2.83 ↑	p < 0.05 Z = -2.51 ↑	p < 0.01 Z = -2.79 ↑	p < 0.01 Z = -4.24 ↑	p < 0.01 Z = -3.99 ↑	p = 0.49 Z = -0.69 ↓	p = 0.99 Z = -0.01 ↑	p < 0.01 Z = -3.82 ↑	p = 0.17 Z = -1.38 ↑	p < 0.05 Z = -2.54 ↑	p < 0.01 Z = -3.34 ↑
SWT	p < 0.01 Z = -4.04 ↓	p = 0.12 Z = -1.56 ↑	p < 0.01 Z = -6.81 ↑	p < 0.01 Z = -7.50 ↑	p < 0.01 Z = -7.41 ↑	p < 0.01 Z = -7.76 ↑	p < 0.01 Z = -4.38 ↑	p < 0.01 Z = -3.49 ↓	p < 0.01 Z = -7.34 ↑	p = 0.11 Z = -1.58 ↑	p = 0.85 Z = -0.20 ↑	p = 0.90 Z = -0.13 ↑
ZXing	p = 0.65 Z = -0.45 ↑	p < 0.01 Z = -2.67 ↑	p = 0.94 Z = -0.07 ↓	p < 0.01 Z = -3.21 ↑	p < 0.01 Z = -2.79 ↑	p = 0.08 Z = -1.76 ↑	p = 0.57 Z = -0.57 ↑	p = 0.40 Z = -0.85 ↑	p = 0.96 Z = -0.05 ↓	p = 0.06 Z = -1.92 ↑	p < 0.05 Z = -2.02 ↑	p < 0.05 Z = -2.05 ↑

Table 3.10. Comparing performance of hybrid and individual methods in terms of TR_{10} using Wilcoxon Signed Rank Test.

System	VSM			LSI			JSM			PMI		
	LSI	JSM	PMI	VSM	JSM	PMI	VSM	LSI	PMI	VSM	LSI	JSM
AspectJ	p = 0.35 Z = -0.94 ↑	p < 0.01 Z = -2.98 ↑	p < 0.01 Z = -7.24 ↑	p = 0.22 Z = -1.23 ↑	p < 0.05 Z = -2.33 ↑	p < 0.01 Z = -7.48 ↑	p = 0.06 Z = -1.88 ↑	p = 0.12 Z = -1.57 ↑	p < 0.01 Z = -6.35 ↑	p = 0.06 Z = -1.92 ↓	p = 0.50 Z = -0.67 ↑	p < 0.01 Z = -3.27 ↓
Eclipse	p < 0.01 Z = -18.26 ↑	p < 0.01 Z = -13.06 ↑	p < 0.01 Z = -24.76 ↑	p = 0.70 Z = -0.38 ↓	p < 0.01 Z = -5.61 ↑	p < 0.01 Z = -20.32 ↑	p < 0.05 Z = -2.10 ↓	p < 0.01 Z = -10.34 ↑	p < 0.01 Z = -19.07 ↑	p = 0.91 Z = -0.12 ↑	p < 0.01 Z = -12.06 ↑	p < 0.01 Z = -2.93 ↑
JodaTime	p = 1.00 Z = 0.00 ↑	p = 0.32 Z = -1.00 ↑	p = 0.06 Z = -1.90 ↑	p = 0.16 Z = -1.41 ↑	p < 0.05 Z = -2.24 ↑	p < 0.01 Z = -2.89 ↑	p = 0.66 Z = -0.45 ↓	p = 0.56 Z = -0.58 ↑	p < 0.05 Z = -2.53 ↑	p = 0.66 Z = -0.45 ↑	p = 0.66 Z = -0.45 ↑	p = 0.18 Z = -1.34 ↑
SWT	p < 0.05 Z = -2.36 ↓	p < 0.05 Z = -2.14 ↑	p < 0.01 Z = -5.75 ↑	p < 0.01 Z = -3.21 ↑	p < 0.01 Z = -3.61 ↑	p < 0.01 Z = -7.62 ↑	p < 0.01 Z = -3.05 ↑	p = 0.20 Z = -1.28 ↓	p < 0.01 Z = -6.25 ↑	p = 0.66 Z = -0.45 ↑	p = 0.21 Z = -1.27 ↑	p = 0.53 Z = -0.63 ↑
ZXing	p = 0.32 Z = -1.00 ↑	p = 0.16 Z = -1.41 ↑	p = 0.56 Z = -0.58 ↓	p = 0.32 Z = -1.00 ↑	p = 0.08 Z = -1.73 ↑	p < 0.05 Z = -2.00 ↑	p = 1.00 Z = 0.00 ↑	p = 0.32 Z = -1.00 ↑	p = 1.00 Z = 0.00 ↓	p = 0.32 Z = -1.00 ↑	p = 0.16 Z = -1.41 ↑	p = 0.16 Z = -1.41 ↑

3.4.3 Discussion

Our analysis has revealed that, on average, when two methods i and j are combined using Eq.3.1, the generated hybrid list is often higher in quality than the lists generated by each method individually. However, the performance gain of the hybrid pair tends to be influenced by the individual performance of i and j. Specifically, the performance gain is:

- Directly proportional to the number of unique, relevant artifacts retrieved by method j, and
- Inversely proportional to the number of relevant files retrieved by the method, i.

The direct proportionality can be explained based on the fact that when two methods retrieve two different sets of artifacts, the hybrid combination has a higher chance of including more relevant artifacts. As a result, we see an increase in performance. Also, if the MRR and the TR_{10} of j are higher than i, the combination method is more likely to have a higher MRR and a higher TR_{10} than i. The second relation, inverse proportionality, can be explained based on the fact that if i has already retrieved most of the relevant artifacts, the chances of j contributing any new relevant artifacts are relatively low. The combination may lose some of the relevant artifacts that i retrieved, but j did not. As a result, we may see a decline in the overall performance of i in terms of MRR and TR_{10}.

To get better insights into these observations, we examine the number of unique links generated by each method. To explain this analysis, consider three different retrieval methods α, β, and γ. Assuming these methods were used to retrieve the code artifacts related to a specific bug report and retrieved the relevant files $\{a, b, c, d, e, f, g, h\}$ as shown in Fig. 3.2. The overlapping areas show the relevant files that were retrieved by multiple methods. In our example, α retrieved $\{a, b, c, d, e\}$. β and γ retrieved $\{d, f, g, h\}$ and $\{e, f\}$ respectively. Given these results, α managed to retrieve 3 unique, relevant artifacts $\{a, b, c\}$ that both β and γ failed to retrieve. α retrieved 4 unique artifacts compared to β and 4 unique artifacts compared γ. β retrieved 3 unique artifacts $\{d, g, h\}$ compared to γ and 3 unique artifacts

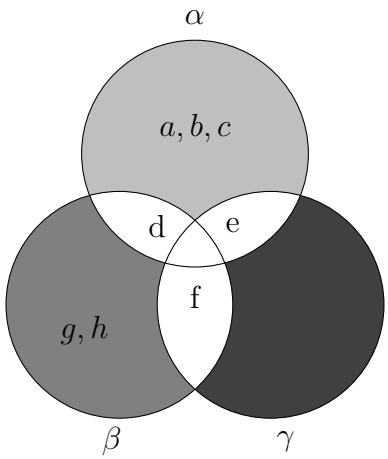


Figure 3.2. Example: An illustration of the calculation of the number of unique artifacts retrieved by three IR methods α, β, and γ.

Table 3.11. The number of unique relevant artifacts (true positives) retrieved by each IR method in comparison to the other methods. The diagonal indicates the number of relevant artifacts unique to the method (were not retrieved by any other method).

	VSM	**LSI**	**JSM**	**PMI**
VSM	93	395	351	794
LSI	794	285	549	679
JSM	855	654	191	550
PMI	1585	1507	1273	1017

$\{f, g, h\}$ compared to α. On the other hand, γ retrieved a unique artifact $\{f\}$ compared to α and a unique artifact $\{e\}$ compared to β. Similar to this example, we repeat this analysis over all of our investigated IR methods. Specifically, we calculate the unique number of artifacts each method retrieved compared to each other method and the number of artifacts that each IR method uniquely captured in all systems. The results are shown in Table 3.11. In general, PMI retrieved the largest and most unique set of artifacts, therefore; it was assigned a larger λ than other methods.

In summary, to answer $\mathbf{RQ_4}$, in most cases, the performance of the hybrid methods has increased at least by a small amount compared to the performance of the individual methods. However, a larger increase (more gain) in the performance is obtained when the combined methods return different sets of relevant artifacts and achieve comparable performances

individually. In addition, combining methods that return a similar set of relevant artifacts does not necessarily increase the performance. Also, combining a high-performing method with a very low-performing method may reduce the performance of the high-performing method, thus, reducing the overall performance of the hybrid method. Finally, to summarize our findings, we revisit our research questions:

- **RQ_1: Is there any global optimal λ that can be used for combining IR methods?** Different hybrid methods require different λ values. This value depends on the individual performance of the combined methods. Specifically, more effective methods should be assigned higher confidence levels. Our analysis also revealed that an average near-optimal λ could be calculated for each hybrid pair.

- **RQ_2: How effective is the hybrid approach compared to the individual IR methods?** Almost all methods experienced a significant performance improvement when combined with other methods using the optimized hybrid approach. This gain was more obvious in larger systems where the number and size of code artifacts were larger, thus resulting in a better performance for a method such as PMI.

- **RQ_3: How effective are the λ-optimized hybrids compared to the unoptimized hybrids?** If optimized correctly, hybrid methods can outperform unoptimized hybrid methods. Specifically, different individual IR methods perform differently; therefore, treating these methods equally in the hybrid combination can limit potential enhancement or, even worse, lead to a decline in the performance.

- **RQ_4: How does the performance of individual IR methods affect the performance of their hybrid pairs?** The performance of hybrid methods depends on the performance of their individual methods. Methods that retrieve more unique links individually (e.g., PMI) bring more value to the hybrid pair, thus should be assigned higher confidence levels.

3.5 Threats to Validity

The analysis presented in this chapter has several limitations that might affect the validity of the results. In this section, we describe our potential internal, external, construct, and conclusion validity threats along with our mitigation strategies.

3.5.1 Internal Validity

Threats to internal validity are influences that can affect the independent variable with respect to causality [125]. A potential threat to the proposed study's internal validity could be made about using a brute-force strategy to calibrate the optimized hybrid method. Our experiment followed an exhaustive search approach to determine average near-optimal values of λ for each IR-method combination. However, since we measured the performance of each combination with different λ values with a 0.1 step size, we believe we captured a near-optimal value of λ that results in the best performance for each combination.

3.5.2 External Validity

Threats to external validity are conditions that limit the ability to generalize the experimental results [125]. In particular, the results of our experiment might not generalize beyond the specific experimental settings used. One potential threat to our external validity is the dataset used in our analysis. In particular, we only experimented with five systems that are limited in size. To mitigate this threat, we compiled our ground-truth from five benchmark software systems that differ in size, the programming language used, and the domain. Using such a diverse dataset can help to enhance the generalizability of our findings. Another threat could be the generalizability of the optimized λ value computed for each hybrid IR-methods. To mitigate the threat, we compiled the λ value from multiple datasets and investigated the performance of the λ over each of the five benchmark datasets. Furthermore, we discussed the value of λ for each IR combination.

3.5.3 Construct Validity

Construct validity is the degree to which the various performance measures accurately capture the concepts they purport to measure [125]. In our experiment, there were minimal

threats to construct validity as the standard performance measures (MAP, MRR, TR_1, TR_5, and TR_{10}), which are commonly used in bug localization research, were used to assess the performance of our investigated methods. As a result, we believe that these measures sufficiently captured and quantified different aspects of performance we were interested in.

3.5.4 Conclusion Validity

Threats to the conclusion validity are concerned with issues that affect the ability to draw the correct conclusion about relations between the treatment and the outcome of an experiment [125]. The main conclusion threat might stem from the low statistical power of the test used to reveal the patterns in the data. In our analysis, we used Wilcoxon signed-rank test. This test is commonly used in bug localization and traceability research to conclude [50, 65, 80, 121, 122]. In these tasks, this test is preferred over parametric tests such as ANOVA as it does not make any assumptions about the normality of the data. Thus no assumptions of the statistical test are violated.

3.6 Conclusion

In this chapter, we investigated the impact of combining different IR methods on bug localization. Our set of IR methods consisted of VSM, LSI, JSM, and PMI combined into hybrid pairs using a λ-optimized method and two unoptimized methods, Borda Count and Score Addition. Five benchmark systems from different application domains were used to conduct our analysis. The performance of the different investigated IR methods was measured using performance indicators that are commonly used in IR-based bug localization research.

Our results showed that combining different methods improved all performance indicators in a statistically significant number of cases. However, the amount of improvement was highly dependent on the performance of individual IR methods. The results also showed that each hybrid pair of IR methods could determine near-optimal confidence levels. Specifically, methods that retrieve more number of unique-relevant artifacts individually should be assigned more confidence when combined with their less effective counterparts; treating the combined IR methods as equal can limit any potential improvement in performance.

Chapter 4
Using IR Methods for BL in OSS

Existing research on IR-enabled bug localization has been focused on traditional proprietary software. Consequently, little effort has been made to assess the effectiveness of such methods in the context of Open Source Software (OSS). To address this limitation, in this chapter, we investigate the effectiveness of IR-enabled BL methods in OSS. In the first phase of our analysis, we propose a systematic process for preparing ground-truth BL datasets for OSS projects. In the second phase, we investigate the performance of several IR methods and their combinations in retrieving bugs in OSS projects. We further propose an approach for enhancing the accuracy of IR methods by exploiting knowledge from previously resolved bug reports. Our analysis is conducted using three large-scale OSS projects. The results show that, in the context of OSS, information-theoretic IR methods significantly outperform other IR methods. Furthermore, combining IR methods enhances the performance of individual methods significantly. The results also show that information from resolved bug reports can significantly enhance the accuracy of BL for new bugs.

4.1 Introduction

Open-Source Software (OSS) is a unique phenomenon in Software Engineering. Unlike traditional commercial software, the idea of OSS is expressed through volunteer participation of programmers who work together to write code, make changes, and maintain the product [39, 131]. In recent years, OSS has become a significant factor in driving the growth of the software industry, with major technological companies, such as Google, Microsoft, turning toward open-source as a more reliable and secure development paradigm [15, 55].

The success of the OSS movement over the past decade can be largely attributed to the emergence of online version control systems, such as GitHub and SourceForge. These platforms provide hosting services to upload, share, maintain code, and establish and manage worldwide social networks of software developers and users. Such platforms also provide the means for the users to report bugs and communicate concerns with OSS developers. However,

this decentralized approach opens up the possibility for issues to be reported by users who lack experience in issue reporting. This often results in developers taking longer to localize and resolve reported bugs [11].

In OSS, a bug's life cycle starts with a user creating a bug report. The bug then goes through different states: Unconfirmed, New, Assigned, Resolved, Verified, and Closed [60]. In a large OSS project, the number of issues reported is typically very high, especially during release. Often, developers have to spend a lot of their time inspecting unproductive bug reports [6]. Recent studies have reported that close to 56% of reported issues in OSS do not help the project [6]. Moreover, as there is a great diversity in the composition of OSS development teams, OSS developers often have a hard time making sense of the reported issue and even communicating with other developers (project contributors) [30]. Consequently, developers assigned to fix a bug have to trace the bug through many massive code modules with limited information.

Our review of existing work on BL in the OSS context revealed a lack of experimental data and research. This lack can be attributed to multiple factors stemming from OSS development's open and diverse nature [6, 30]. Bug reports in OSS are typically created by users with a wide range of expertise [11]. In addition, contributors involved in OSS projects are often of varying expertise and come from different backgrounds [30]. Therefore, bugs are usually worked on by different developers throughout long periods and over multiple project releases. This makes preparing ground-truth datasets for IR-enabled BL research in OSS a very challenging process. To overcome these limitations, in this chapter, we introduce a simple systematic procedure for automatically devising ground-truth bug localization datasets for open-source projects. We then evaluate the performance of multiple IR methods and their combinations in localizing bugs in OSS. Finally, we propose and evaluate a novel method to improve the performance of our IR-enabled BL methods using knowledge extracted from previously resolved bug reports.

4.2 Data and Oracle

In OSS management platforms, such as Github, a user can add an *issue* to report a bug, request a feature change, or get information from developers. Whenever a new issue is added, the issue is automatically set to *open*. Each issue consists of a single-line title and a detailed description. An issue can also be associated with one or more *labels*. Labels help developers to organize and prioritize their work. They are also used to signify priority, category, or other information that developers may find useful. When an issue is resolved, a developer or the project manager closes it and sets it to *closed*. A valid issue is typically resolved by making a code change or providing a suggested resolution in the comments. Fig. 4.1 shows a sample issue from the Kubernetes project. When a code change is made for an issue, Github keeps track of the changes made. These changes are retained in Github as a *commit* or a *reference*. Commits and references can be described as follows:

- **Commit**: In version control systems, a commit is a permanent record of some change made to the source code. Once a commit is created, it is stored indefinitely in the repository. Each commit is assigned a unique identifier called the *hash* generated by a cryptographic hash function. In addition to the changes made to the source code, the commit records the changed files, developer information, and the timestamps of the different changes. Thus, at any given time, a developer can retrieve the latest committed version of the code or a version associated with a specific commit.

- **Reference**: Github provides various means to associate code changes to an issue. One way is to add commit references to an issue. Through these references, code contributors can associate the code changes made to resolve an issue, in the form of commits, to the issue.

To investigate the performance of IR methods in localizing bugs in OSS, we create a ground-truth oracle. TThe oracle consists of closed issues and the source code files that have been changed to fix each issue. However, since we deal with version control systems, not

Portworx volume driver is dropping namespace information when creating a PVC #59606

BUG REPORT: During the create workflow, Portworx native driver has the PVC namespace information. But this isn't getting passed to the Portworx Create API. As a result, the Portworx service side handler doesn't get the namespace information.

It will also be helpful to send PVC annotations in the Create API call.

What happened:
Portworx native driver didn't send PVC namespace to the Portworx Create API

What you expected to happen:
Portworx native driver should send the PVC namespace to the Portworx Create API

Figure 4.1. Issue #59606 from the Kubernetes project.

all bugs are fixed within the same release. Therefore, the exact code version where the bug is fixed must first be determined to create a valid oracle. Then, using commits, references, and issue reports, we extract such information has to be extracted from the repository. This process can be described as follows:

- **Retrieving bug reports**: The repository issues are first downloaded using GitHub's API. Then, we extract its title, labels, description, and references for each issue. Other commits or code files referenced in each commit are also extracted.

- **Retrieving Source Code**: Bug localization for an issue is carried out on the version of the source code where the bug still exists and needs fixing. To retrieve this version, we use the commits associated with the issue. First, we clone the latest version of the source code. *Cloning* refers to retrieving the latest repository and version histories from Github. We then checkout the repository version linked to the commit associated with the issue. The *checkout* process changes the source code version to the version where the commit was made. We then rollback the commit changes to remove the changes made to fix the bug from the source code. *Rollback* is the process of reverting the changes made in a specific commit, in our case, the changes that fixed the issue. Finally, at the end of the process, for each issue, we have a version of the source code that contains the bug indicated in the bug report. Fig. 4.2 shows the process of retrieving the buggy source code version from the latest version of source code.

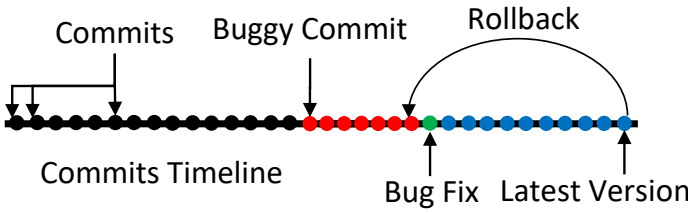


Figure 4.2. The process of retrieving a buggy copy of the source code using a rollback.

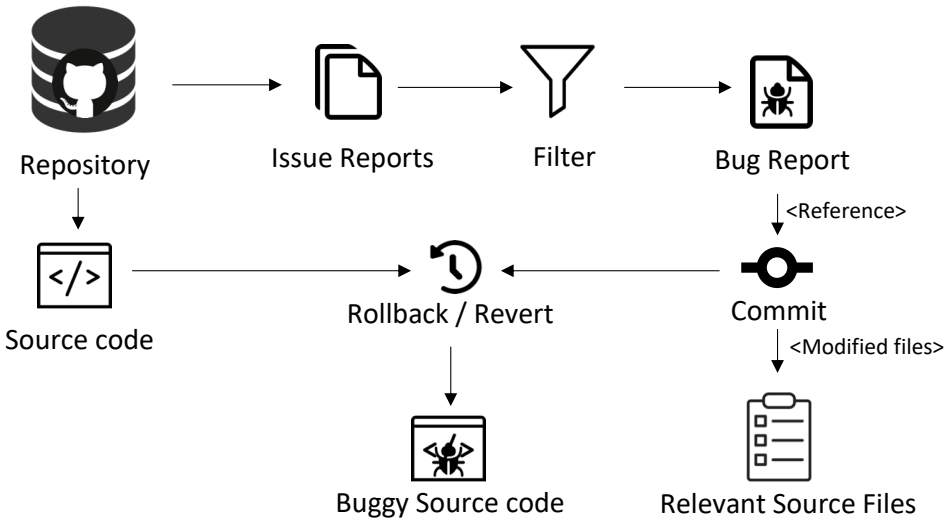


Figure 4.3. The process of creating a ground-truth OSS bug localization dataset.

- **Creating the Ground-Truth**: For our analysis, a bug's relevant file list is required to verify the results of our bug localization methods. We obtain this information from the list of files associated with the commits referenced in the reported issue.

Fig. 4.3 summarizes the proposed process for creating an oracle for bug localization tasks in OSS from version control systems. At the end of the process, for each issue, we have *a)* the bug report title and description or the bug query, *b)* the latest version of source code that contains the bug indicated in the bug report, and *c)* the list of files which were modified to fix the bug.

4.3 Experimental Setup

In this section, we describe our experimental subject projects and our main research questions.

4.3.1 Experimental Datasets

We created our experimental dataset using the oracle generation process described in Section 4.2. We selected three large-scale projects from GitHub to generate our ground truth. These projects are:

- **Kubernetes**: Kubernetes is a production-grade open-source platform for deployment, scaling, and managing containerized applications and services. The project is active, with contributors constantly applying changes to the project based on daily new issues. The system is primarily developed using the *go* programming language. The project is fairly large, with about 18,000 files, of which about 11,382 files include *go* source codes. The project has more than 76,968 commits, 500 releases from about 2,092 contributors, and about 29,687 issues (open and closed) logged.

- **CoreCLR**: .Net Core (CoreCLR) is a general-purpose development platform maintained by Microsoft and the .NET community on GitHub. CoreCLR is cross-platform supporting Windows, macOS, and Linux and can be used to build devices, cloud, and IoT applications. The project is primarily written in C# and C++ and contains about 29,800 files, of which close to 21,850 files contain C++ or C# source codes. The Github project for CoreCLR has more than 20,736 commits and 81 releases from about 556 contributors and close to 8,512 open and closed issues.

- **Angular.js**: Angular.js is a structural framework for dynamic web applications. The source code for this framework is also hosted in Github and is primarily written in JavaScript. The project contains close to 1,900 files, of which 1,103 are JavaScript files. The project has 1,596 code contributors, 204 releases, and about 8,961 commits. So far, about 8,932 open and closed issues have been logged.

For our experiment, we only use issue reports that project developers manually categorize as bugs. We identify these reports by examining the labels associated with each report in each project. These labels are listed in Table 4.1. Additionally, only issues associated with

Table 4.1. The OSS projects used as experimental subjects in our analysis.

Project	Stars	Commits	Releases	Contributors	Issues Reported	Source File	Lang.	Bug Label	Bug Report
Kubernetes	50925	76968	500	2092	29687	11382	Go	kind/bug	140
CoreCLR	11235	20736	81	556	8512	8860	C#, C++	test bug, bug	299
Angular.js	59489	8961	204	1596	8932	1103	JavaScript	type: bug	445

at least one commit are selected for our experiment as commits contain information about the list of files associated with each bug. After applying the label and commit filters on the last release of each of our subject projects, we ended up with 140, 299, and 445 bug reports for Kubernetes, CoreCLR, and Angular.js, respectively. A description of our dataset and subject projects is shown in Table 4.1.

4.3.2 Research Questions

The main objective of our experimental analysis is to evaluate the performance of various IR-enabled bug localization methods in open-source projects and investigate our proposed performance improvement method. To guide our analysis, we formulate the following research questions:

- **RQ$_1$**: How effective are classical IR methods in localizing bugs in OSS?
- **RQ$_2$**: Does combining IR methods improve the accuracy of bug localization in OSS?
- **RQ$_3$**: Can previously fixed bugs help to localize newly reported bugs?

4.4 Results

In this section, we examine the performance of individual (RQ$_1$) and hybrid (RQ$_2$) IR methods in localizing bugs in our subject OSS projects. These methods and the evaluation metrics used to assess the performance of our different retrieval configurations are described in Chapter 2 and Chapter 3.

4.4.1 Individual Methods Performance

We start our analysis by examining the performance of individual IR methods. Each method is used to retrieve the code files relevant to each bug report in our experimental

projects. Table 4.5[1] shows the TR_1, TR_5, and TR_{10} values for each method for each dataset, and Fig. 4.4 shows the MAP and MRR. We used Wilcoxon signed-rank test to detect statistical significance in the loss/gain in MAP and MRR. To test for statistical significance in terms of TR, we use McNemar's test [41]. The results are shown in Table 4.6.

The results show that TSS_{PMI} outperforms both VSM and LSI in all three datasets. This can be attributed to the fact that, unlike a method such as VSM, TSS_{PMI} does not rely on exact word matching. This gives this method an advantage in the context of OSS, where bug reports are created by a diverse set of users and developers. Moreover, the results also show that the topic modeling method LSI performs poorly compared to other methods. This can be explained based on the fact that a single bug is typically related to multiple source code files. Therefore, a topic cluster created for representing a bug report may not adequately match the topic clusters created for its relevant source files. The information-theoretic method TSS_{PMI} overcomes these problems by matching bug reports and source code based on the relatedness of their constituent words, thus, overcoming the vocabulary mismatch problem affecting OSS projects.

Regarding the differences in performance between the different projects, the better performance of VSM in Angular.js can be attributed to the fact that this project includes a small number of large files compared to the other projects. As a result, bug reports are typically unique and only related to a few source files. In the larger projects, Kubernetes and CoreCLR, VSM, and LSI perform poorly compared to TSS_{PMI}. These results attest that bug reports and source files containing the buggy source codes may not share the exact words (i.e., bug reports are expressed using informal language). Our statistical testing shows that the difference in performance between the IR methods in these two projects is statistically significant except between VSM and LSI in Kubernetes. These observations answer our $\mathbf{RQ_1}$: information-theoretic IR methods, such as TSS_{PMI}, can be more suited for bug localization in OSS.

[1]Tables are moved to the end of the chapter for readability purposes.

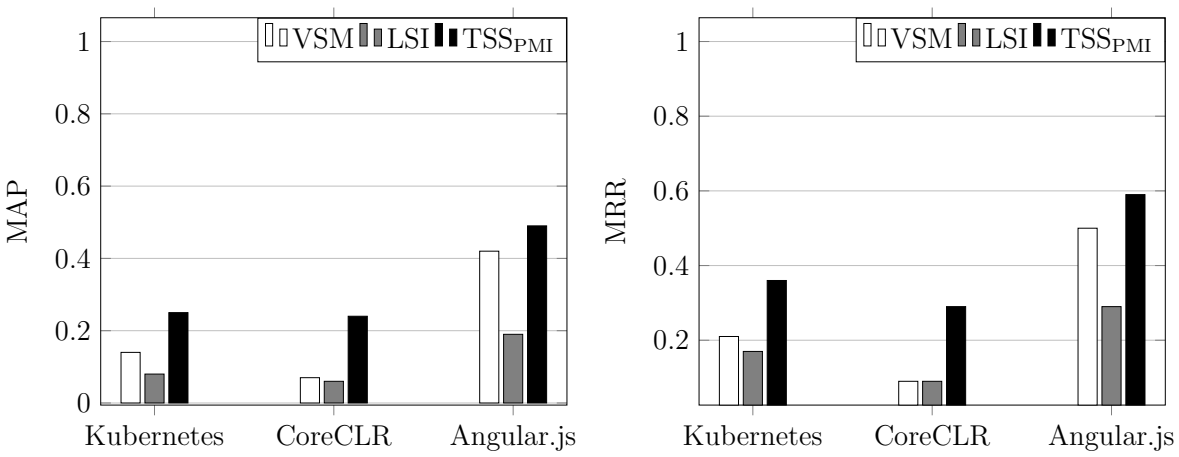


Figure 4.4. The performance of VSM, LSI, and TSS_{PMI} in terms of MAP and MRR in Kubernetes, CoreCLR, and Angular.js

4.4.2 Hybrid Methods Performance

Table 4.7 shows the results of the combination of different IR methods using Score Addition, Borda Count, and the λ-optimized approach. In general, the results came consistent with our previous findings; the λ-optimized approach performs better than Score Addition and Borda Count. The performance improvement of this specific approach over the uncalibrated approaches can be attributed to the fact that it takes the performance of the constituent IR methods into consideration and adjusts their contribution accordingly. In particular, using the λ-optimized approach, of the two IR methods used in the combination, the method that performs better is assigned higher confidence (λ), while Score Addition and Borda Count treat constituent IR methods equally.

Table 4.8 shows the improvement in the performance of the hybrid methods over individual methods. In all projects, the performance of VSM and LSI increased when combined with PMI. However, the performance of PMI decreased in Angular.js and CoreCLR when combined with LSI. Similarly, the performance of PMI decreased when combined with VSM in CoreCLR. This decline can be attributed to VSM and LSI performing relatively poorly in these projects. However, in all other cases, both methods benefited from the combination.

To detect statistical significance in the loss/gain in MAP and MRR, we use Wilcoxon signed-rank test and McNemr's test for TR_{10}. The results are shown in Table 4.9 (only

λ-optimized combination's result included). In general, to answer $\mathbf{RQ_2}$, using an optimized method for combining IR methods can significantly improve performance.

4.5 Enhanced IR-Enabled BL for OSS

Large software systems have a rich development and maintenance history. Such information is indispensable for developers [27, 68, 106, 138]. Developers frequently ask questions about the history of the code to understand the change rationale, track bugs to their origin, or trace features to their older versions. During software evolution, code change is typically influenced by maintenance activities. Therefore, exploiting such knowledge is expected to provide important information about current unresolved bugs. Based on these observations, in this section, we propose a novel method to improve the performance of IR-enabled bug localization methods in OSS using information about bugs that have already been fixed.

4.5.1 Method Description

In our approach, we use previous bug reports and their associated modified relevant files to provide cues to new, unresolved bug reports. The proposed approach is applied to improving the performance of TSS_{PMI} as this particular IR method managed to achieve the best retrieval accuracy in the context of OSS software, significantly outperforming other IR methods in localizing bugs. As explained in Section 2.2.3, TSS_{PMI} estimates the semantic similarity between a bug report and a source code file by calculating the pairwise semantic similarity of their words. Our objective is to feed TSS_{PMI} with more information to make more accurate estimations of semantic similarity, thus, enhancing the overall accuracy of bug localization. Formally, the enhanced equation of TSS_{PMI} between a bug report (b) and a code file (d_i) can be described as follows:

$$TSS^{A}_{PMI}(b, d_i) = TSS_{PMI}(b, d_i) + \alpha(b, d_i) \tag{4.1}$$

where $TSS_{PMI}(b, d_i)$ is the TSS_{PMI} score (Eq. 2.10) between the bug report (b) and the code file (d_i). $\alpha(b, d_i)$ is an adjustment value calculated using previously fixed bug reports. Formally, $\alpha(b, d_i)$ can be calculated as:

Using these three values, $\alpha(b, d_i)$ can be written as:

$$\alpha(b, d_i) = \frac{1}{n} \times \sum_{x=1}^{n} \frac{\log(1 + TSS_{PMI}(b, b_x) \times TSS_{PMI}(b_x, d_i))}{I(b_x, d_y)} \times R(b_x, d_i) \qquad (4.2)$$

In the above equation, n is the number of the closed bug reports in the project, and the other terms can be described as follows:

- **Bug report similarity**: Bug report similarity, $TSS_{PMI}(b, b_x)$, is measured as the text semantic similarity between the bug report (b) and the past bug reports (b_x). Similar bug reports tend to have similar relevant source code files. Therefore, similar bug reports ($TSS_{PMI}(b, b_x) > 0$) should have positive influence in updating $TSS_{PMI}(b, d_i)$. On the other hand, dissimilar bug reports ($TSS_{PMI}(b, b_x) < 0$) should negatively influence α.

- **File relevancy**: Represented as $R(b_x, d_i)$, file relevancy indicates if the source file d_i is relevant to the bug report b_x. $R(b_x, d_i)$ assumes a value of 0 if d_i is not relevant to b_x, and 1 if d_i is relevant (changed to resolved) to b_x.

- **File ranked list position**: Represented as $I(b_x, d_i)$, The ranked list position of a source file d_i refers to the position of d_i in the ranked list generated when localizing the bug report b_x. A higher value (lower rank) of $I(b_x, d_i)$ indicates less similarity between the bug report (b_x) and the source code file (d_i) and vice versa.

To scale down the value of $\alpha(b, d_i)$, we take the *log* of the score. The value of $TSS_{PMI}(b, b_x)$ and the value of $TSS_{PMI}(b_x, d_i)$ can range from -1 to 1, and because the *log* does not operate with negative numbers, we add 1 to the numerator so that the possible values for the *log* range from 0 to 2.

Example: To explain the operation and the performance improvement of TSS_{PMI} using Eq. 4.1, consider the retrieval scenario in Table 4.2 for a software system with 10 code artifacts (d_1, d_2, ..., d_{10}) and three bug reports (b_1, b_2, and b). TSS_{PMI} is used to calculate

the similarity score between the code artifacts and the bug reports b_1 and b_2. The bug b_1 has 3 related source artifacts, d_8, d_5, and d_9 ranked 2, 4, and 8 in the list, and the bug b_2 has 2 related code artifacts, d_{10} and d_2 ranked 3 and 5 in the list.

The localized bug b, has 3 related source artifacts, d_8, d_9, and d_6 ranked 3, 5, and 6 respectively in the list. Assuming $TSS_{PMI}(b, b_1) = 0.5$ and $TSS_{PMI}(b, b_2) = -0.5$, the modified score $TSS^A_{PMI}(b, d_8)$ can be calculated as follows:

$$
\begin{aligned}
TSS^A_{PMI}(d_8) = \frac{1}{n} \times (&\frac{\log(1 + TSS_{PMI}(b, b_1) \times TSS_{PMI}(b_1, d_8))}{I(b_1, d_8)} \times R(b_1, d_8) \\
&+ \frac{\log(1 + TSS_{PMI}(b, b_2) \times TSS_{PMI}(b_2, d_8))}{I(b_2\, d_8)} \times R(b_2, d_8)) \\
&+ TSS_{PMI}(b, d_8) \\
= \frac{1}{1} \times (&\frac{\log(1 + 0.5 \times 0.760)}{2} \times 1 + \frac{\log(1 + -0.5 \times 0.776)}{1} \times 0) + 0.843 \\
= 0.07 &+ 0 + 0.843 \\
= 0.913 &
\end{aligned}
$$

Similarly, the new TSS^A_{PMI} scores for other source files are also calculated as shown in Table 4.2. The *Change* column indicates the change in the score. The results in Table 4.2 show that the ranking for all 3 relevant files increased, increasing the MAP from 0.41 to 0.76 and MRR from 0.33 to 1 for the bug report b.

4.5.2 Analysis of TSS$^A_{\text{PMI}}$ Performance

Table 4.3 shows the results of using TSS_{PMI} and $\text{TSS}^A_{\text{PMI}}$ in terms of TR_1, TR_5, and TR_{10} in our subject projects and Fig. 4.5 shows the MAP and MRR results. In general, $\text{TSS}^A_{\text{PMI}}$ outperforms TSS_{PMI} for all evaluation metrics for all projects. On average, the performance of $\text{TSS}^A_{\text{PMI}}$ is higher by 135.14%, 26.90%, and 16.81% for TR_1, TR_5, and TR_{10}, respectively, compared to TSS_{PMI}. Furthermore, the performance of $\text{TSS}^A_{\text{PMI}}$ is higher by 79.16% and 68.55% for MAP and MRR, respectively, when compared to TSS_{PMI}.

We used Wilcoxon signed-rank test to test for statistical significance in the MRR and MAP results and McNemar's test for statistical significance in TR. The results in Table 4.4

Table 4.2. Example: Improving the bug localization performance of a bug report b using bug localization results from two previous bug reports: b_1 and b_2. Relevant code artifacts are shown in gray color.

Rank	$\mathbf{b_1}$		$\mathbf{b_2}$		$\mathbf{b}$		$\mathbf{b}^A$		
	Source	Similarity	Source	Similarity	Source	Similarity	Source	Similarity	Change
1	d_6	0.769	d_8	0.776	d_4	0.859	d_8	0.913	↑
2	d_8	0.760	d_5	0.769	d_2	0.846	d_4	0.859	—
3	d_{10}	0.759	d_{10}	0.768	d_8	0.843	d_9	0.852	↑
4	d_5	0.754	d_9	0.764	d_{10}	0.841	d_5	0.836	↑
5	d_4	0.750	d_2	0.763	d_9	0.835	d_6	0.834	—
6	d_1	0.739	d_1	0.763	d_6	0.834	d_1	0.810	—
7	d_2	0.738	d_7	0.757	d_1	0.810	d_2	0.804	↓
8	d_9	0.737	d_3	0.757	d_7	0.802	d_7	0.802	—
9	d_7	0.735	d_4	0.750	d_5	0.801	d_3	0.794	—
10	d_3	0.725	d_6	0.747	d_3	0.794	d_{10}	0.773	↓

show that in all projects, $\text{TSS}^A_{\text{PMI}}$ experienced significant improvement in MAP and MRR when compared to TSS_{PMI}. The statistically significant improved performance of $\text{TSS}^A_{\text{PMI}}$ can be attributed to the fact that using information from previously fixed bugs adjusts the similarity score of both relevant and irrelevant source files for the new bug report being localized. The adjustment is positive for relevant files when the bug reports are similar, and negative for the irrelevant files when the bug reports are dissimilar. In other words, if the bug reports are similar past bug reports increase the score of the relevant files. If the bug reports are dissimilar, past bug reports decrease irrelevant files' similarity scores. For example, in Angular.js, the increase in TR_5 and TR_{10} values were not significant. This comes from the observation that Angular.js has fewer and larger size files. As a result, TSS_{PMI} lists relevant files in the top 5 and the top 10 positions for most queries. Therefore, when we compare the performance of $\text{TSS}^A_{\text{PMI}}$ to TSS_{PMI}, we do not see a significant improvement for TR_5 and TR_{10}.

Fig. 4.6 shows the performance gain in the Average Precision (AP) values and the Reciprocal Rank (RR) values of $\text{TSS}^A_{\text{PMI}}$ over TSS_{PMI} for each bug report chronologically

Table 4.3. The performance of TSS_{PMI} and TSS^{A}_{PMI} in terms of TR_1, TR_5, and TR_{10}.

Project	Method	TR_1(%)	TR_5(%)	TR_{10}(%)
Kubernetes	TSS_{PMI}	31.21	40.26	44.63
	TSS^{A}_{PMI}	45.23	50.60	55.64
CoreCLR	TSS_{PMI}	14.88	46.05	59.53
	TSS^{A}_{PMI}	58.60	67.44	74.88
Angular.js	TSS_{PMI}	45.00	75.00	85.00
	TSS^{A}_{PMI}	75.00	81.43	85.00

(*x-axis*). For each report, the change in AP and RR (*y-axis*) is calculated as:

$$\frac{AP^{A}_{PMI} - AP_{PMI}}{AP_{PMI}}, \quad \frac{RR^{A}_{PMI} - RR_{PMI}}{RR_{PMI}}$$

The results show that, for Kubernetes and Coreclr, the performance improvement using TSS^{A}_{PMI} is greater for newer issues than older ones. In other words, the performance improvement of TSS^{A}_{PMI} over TSS_{PMI} increase over time. This indicates that using more past bug reports (more historical information) in computing TSS^{A}_{PMI} improves its performance. For example, for the 100^{th} bug report in Kubernetes, the AP of TSS^{A}_{PMI} is approximately 75% greater than the AP of TSS_{PMI}, whereas the AP of TSS^{A}_{PMI} for the 200^{th} bug report is approximately 106% greater than the AP of TSS_{PMI}. In general, as more historical information is used, the similarity scores between new bug reports and source code files are adjusted multiple times. This helps source files related to the bug (true positives) to climb up the list while false positives are pushed down. In Angular.js, using more past bug reports does not necessarily enhance the performance at the same rate as other projects. This can be attributed to the already high performance of TSS_{PMI} in Angular.js. As seen in Table 4.3, TSS_{PMI} had successfully localized relevant files in the top 10 positions of relevant files for 85% of the bugs. There is only a little room for improvement for TSS^{A}_{PMI} to make.

4.6 Discussion

In this section, we summarize our main findings and discuss their implications.

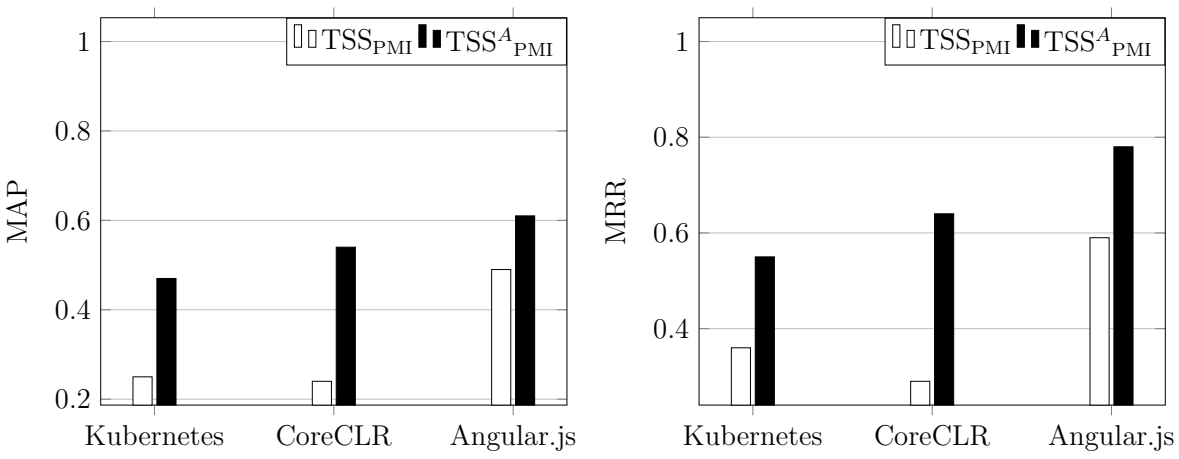


Figure 4.5. The performance measures of $\mathrm{TSS}_{\mathrm{PMI}}$ and $\mathrm{TSS}^{A}{}_{\mathrm{PMI}}$ in terms of MAP and MRR

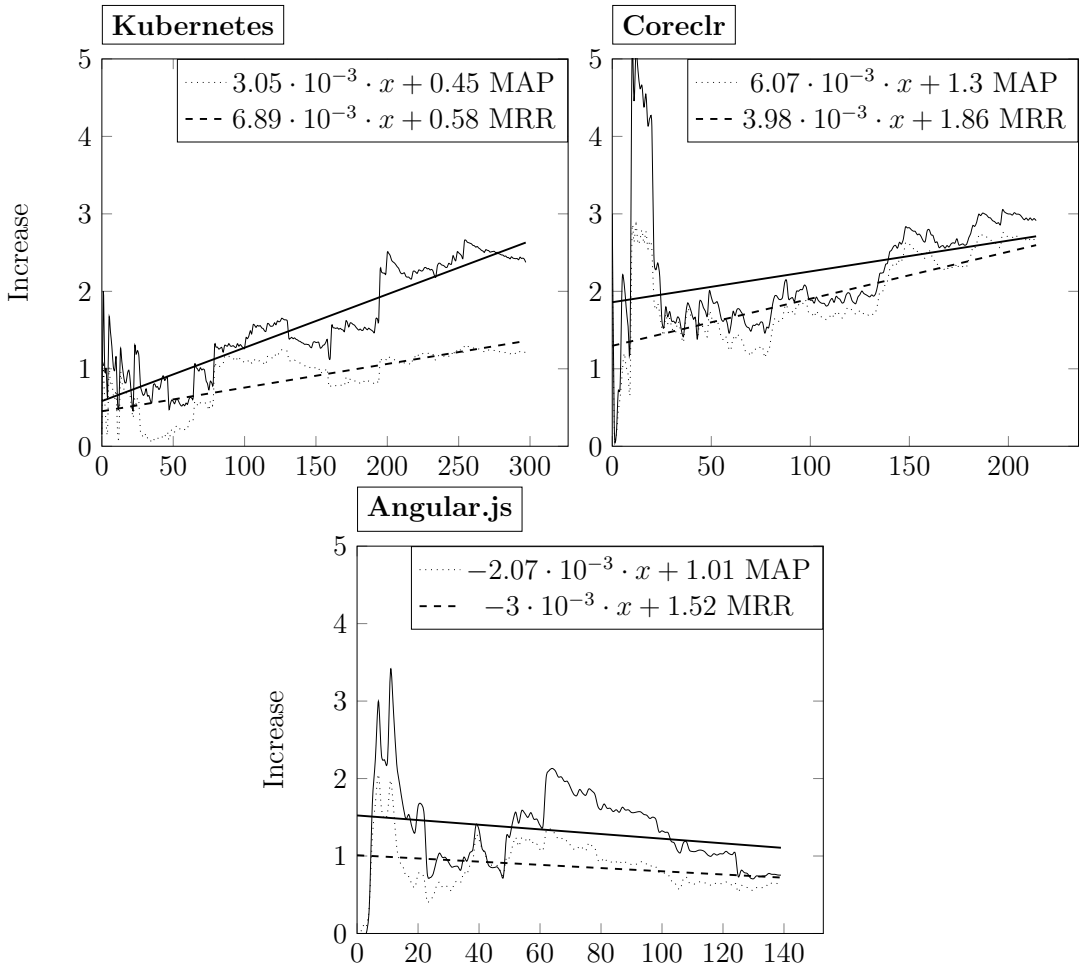


Figure 4.6. Differences in the MAP values and the MRR values when using $\mathrm{TSS}^{A}{}_{\mathrm{PMI}}$ compared to $\mathrm{TSS}_{\mathrm{PMI}}$ with time.

Table 4.4. Comparing the performance of the TSS_{PMI} with TSS^{A}_{PMI} in terms of RR and AP using Wilcoxon Signed Rank Test and TR_1, TR_5, and TR_{10} using McNemar's test.

Project	Wilcoxon Signed Rank Test		McNemar's test		
	MAP	MRR	TR_1	TR_5	TR_{10}
Kubernetes	**-6.857, <0.05**	**-7.248, <0.05**	**0.000**	**0.001**	**0.012**
CoreCLR	**-10.920, <0.05**	**-10.675, <0.05**	**0.000**	**0.000**	**0.000**
Angular.js	**-4.835, <0.05**	**-4.718, <0.05**	**0.000**	0.110	0.789

4.6.1 Summary of Findings

To summarize our findings in this Chapter, we revisit our research questions:

- **RQ_1**: How effective are classical IR methods in localizing bugs in OSS? Our results show that IR methods localize bugs in OSS. TSS_{PMI} significantly outperforms VSM, JSM, and LSI in MAP, MRR, and TR in OSS.

- **RQ_2**: Does combining IR methods improve the accuracy of bug localization in OSS? Our results show that in the context of OSS, all IR methods experienced significant improvement in all performance indicators when combined with other methods. In addition, the gain for each IR method was clearer for larger systems; however, the best performing method (TSS_{PMI}) showed a small decline in performance when combined with a method such as LSI, which had the worst performance of all other methods.

- **RQ_3**: Can previously fixed bugs help to localize newly reported bugs? Our results show that using information from previously fixed bug reports can significantly improve the performance of individual IR methods, particularly TSS_{PMI}, in terms of MAP, MRR, and TR values. In addition, the relevant files for past bug reports can be used to adjust the similarity scores of source code artifacts based on the semantic similarity of the new bug report to past bug reports. The results also showed that the performance gain increased as more past bug reports were used.

4.6.2 Using IR-enabled BL Methods in OSS

Responding to bug reports is a crucial task for software projects as it affects the project's quality, reputation, and long-term success. This process becomes particularly difficult for

open source software, where inexperienced contributors are often quite large [133]. For example, by 2013, out of 64,000 bugs reported for Mozilla Firefox, 78% of the bug reports were either duplicate, invalid, or incomplete [133]. Therefore, for a more efficient bug localization in OSS, automated methods must be able to analyze a large number of issues effectively and achieve higher accuracy in locating the buggy files.

Our study has shown that classical IR approaches, such as VSM and JSM, are not optimal for bug localization in the context of OSS. This can be mainly attributed to the vocabulary mismatch problem affecting OSS bug reports, and incomplete bug descriptions are quite common in OSS projects [62]. Davies et al. [32] pointed out a clear mismatch between the information that typically appears in bug reports and the information that developers would wish to appear, reporting that about 79% of the bug reports in the data they analyzed contained a description that was shorter than 100 words [32]. On the other hand, more semantically aware approaches, such as LDA and LSI, while they can outperform a basic method such as VSM, can be computationally expensive [132].

In our analysis, we proposed TSS_{PMI} as an efficient alternative for retrieving buggy source files in OSS with minimal computational cost. The higher accuracy of TSS_{PMI} stems from the fact that PMI calculates the similarity between words based on their usage in the entire system, and not necessarily only in the bug report or the source code files being examined. Therefore, TSS_{PMI} is also accompanied by a lower computational cost. Furthermore, once the values of PMI between all words in the system are computed, computationally, TSS_{PMI} becomes a simple dictionary lookup of PMI values to compute the similarities between a bug report and source code.

4.6.3 Improving the Performance of BL Methods

Several techniques have been proposed in the literature to improve the performance of BL methods. These techniques primarily use stack trace information [88, 104, 123] or machine (deep) learning [94, 67, 129] to improve the retrieval accuracy of buggy code. However, in the context of OSS, such techniques often encounter limitations. For instance, stack traces

are not always readily available to the IR method. In a study by Bettenburg et al. [11], the authors reported stack traces as one of the most difficult pieces of information to provide. The availability of stack trace becomes even harder when users of different levels of expertise and involvement in the project are involved in reporting issues [133]. If the bug report does not provide enough information about the source code, which is usually the case for OSS [32], the performance of bug localization is adversely affected.

Machine learning methods, especially deep learning, are becoming increasingly popular in software engineering [38]. These techniques are primarily preferred to overcome the lexical gap between source code and bug reports [94]. However, the main limitation of such techniques is the computation cost associated with training and tuning generated models. In a study by Polisetty et al. [94], the authors reported that using deep learning in BL requires large amounts of data along with hardware infrastructure (such as GPUs and memory) to work, which can be both challenging and expensive for mainstream software practitioners, especially OSS developers who are only partially involved in the projects.

To work around these limitations, in our experiment, we used past bug reports to enhance the performance of bug localization methods. Using historical data for improving bug localization has shown promising results in the literature [112, 123]. However, compared to existing methods, our method presents several advantages, such as the fact that no information (stack traces) is needed and does not require as much computational overhead.

4.6.4 BL Datasets for OSS

Our literature analysis for bug localization showed a lack of experimental datasets for researchers to conduct experiments. In particular, most experiments are conducted with the same set of bug localization datasets. These datasets mostly included the bugs from Firefox [120], Eclipse [94, 104, 120, 123, 137], SWT [94, 104, 123, 137], ZXing [104, 123, 137] and AspectJ [94, 104, 123, 137]. We believe that using older datasets for experimentation fails to capture the new trends in software evolution, new programming languages, new methodologies in bug reporting, and the increasing complexity of software systems. We

believe that our proposed procedure for creating new datasets for IR-enabled BL in OSS will allow researchers to conduct more valid experiments using more up-to-date data.

4.7 Conclusions

In this chapter, we investigated the performance of different IR bug localization methods in the context of OSS. Our set of IR methods consisted of VSM, LSI, TSS_{PMI} and their hybrid combinations. Furthermore, we introduced a new method to improve the retrieval accuracy of IR methods using information extracted from previously resolved bug reports.

To conduct our analysis, we introduced a systematic approach for devising ground-truth datasets for OSS systems. Our analysis was conducted on a dataset prepared from three large-scale OSS projects. Furthermore, bug localization performance was measured using conventional performance indicators commonly used in IR-based bug localization research. These measures included MAP, MRR, TR_1, TR_5, and TR_{10}. In addition, Wilcoxon signed-rank test and McNemar test were used to test for statistical significance. Our results showed that TSS_{PMI} outperformed other IR methods for bug localization in all projects. Furthermore, combining IR methods into hybrid pairs improved all performance indicators except for a few cases. Our results also showed that using historical information from previously resolved bug reports can significantly improve the performance of TSS_{PMI}.

Table 4.5. The performance of the individual IR methods in terms of TR_1, TR_5, and TR_{10}.

Project	Method	Top_1(%)	Top_5(%)	Top_{10}(%)
Kubernetes	VSM	12.71	27.76	38.13
	LSI	12.08	21.81	26.17
	TSS_{PMI}	31.21	40.29	44.63
CoreCLR	VSM	4.65	11.16	15.35
	LSI	4.65	12.10	15.35
	TSS_{PMI}	14.88	46.04	59.53
Angular.js	VSM	37.14	67.85	78.57
	LSI	19.28	40.00	45.71
	TSS_{PMI}	45.00	75.00	85.00

Table 4.6. Comparing the performance of individual IR methods in terms of RR and AP using Wilcoxon Signed Rank Test, and TR_{10} using McNemar's test.

Project	VSM			LSI			PMI		
	MAP	**MRR**	**TR_{10}**	**MAP**	**MRR**	**TR_{10}**	**MAP**	**MRR**	**TR_{10}**
Kubernetes	**-7.895, 0.000**	**-4.204, 0.000**	**0.000**	-0.778, 0.437	-0.333, 0.739	0.525	**-6.633, 0.000**	**-3.572, 0.000**	**0.001**
CoreCLR	**-10.051, 0.000**	**-9.641, 0.000**	**0.000**	**-10.137, 0.000**	**-9.885, 0.000**	**0.000**	**-4.607, 0.000**	**-3.016, 0.000**	1.000
Angular.js	**-7.498, 0.000**	**-6.169, 0.000**	**0.000**	**-2.348, 0.019**	**-2.114, 0.035**	0.124	**-7.449, 0.000**	**-5.971, 0.000**	**0.000**

Table 4.7. The performance of the hybrid IR methods in terms of MAP, MRR, TR_1, TR_5, and TR_{10}.

Project	**Method**	**Score Addition**					**Borda Count**					**λ-optimized**				
		MAP	**MRR**	**TR_1**	**TR_5**	**TR_{10}**	**MAP**	**MRR**	**TR_1**	**TR_5**	**TR_{10}**	**MAP**	**MRR**	**TR_1**	**TR_5**	**TR_{10}**
Kubernetes	VSM-LSI	0.19	0.31	22.15	39.26	51.01	0.17	0.29	20.47	38.25	49.00	0.19	0.31	22.15	39.26	51.01
	VSM-PMI	0.19	0.27	17.45	36.23	48.86	0.24	0.34	22.48	47.23	57.13	0.24	0.35	21.13	47.21	57.37
	LSI-PMI	0.14	0.26	18.11	31.54	39.79	0.19	0.34	24.82	41.27	51.90	0.24	0.36	25.82	48.64	58.71
CoreCLR	VSM-LSI	0.10	0.14	8.37	19.07	23.72	0.10	0.14	9.3	19.53	22.79	0.10	0.14	8.37	19.07	23.72
	VSM-PMI	0.13	0.15	7.9	21.86	31.62	0.19	0.23	13.95	35.81	43.26	0.22	0.24	13.02	33.49	46.98
	LSI-PMI	0.13	0.17	9.77	21.86	26.05	0.19	0.26	18.14	34.42	41.86	0.27	0.33	21.40	44.19	56.28
Angular.js	VSM-LSI	0.33	0.48	32.86	55.71	67.86	0.30	0.45	32.86	59.29	66.42	0.33	0.45	32.86	55.71	67.86
	VSM-PMI	0.72	0.55	40.00	73.57	87.14	0.53	0.62	47.86	82.14	92.14	0.82	0.63	48.57	81.43	92.14
	LSI-PMI	0.26	0.39	28.57	48.57	57.14	0.37	0.56	45.00	68.57	75.00	0.39	0.53	41.43	67.14	78.57

Table 4.8. The performance gain of the hybrid IR methods in MAP, MRR, TR_1, TR_5, and TR_{10} compared to the performance of individual IR methods.

Project	Combination	Individual	Score Addition					Borda Count					λ-optimized				
			MAP	MRR	TR_1	TR_5	TR_{10}	MAP	MRR	TR_1	TR_5	TR_{10}	MAP	MRR	TR_1	TR_5	TR_{10}
Kubernetes	VSM-LSI	VSM	**38.17**	**48.15**	**74.27**	**41.44**	**33.78**	**20.74**	**40.71**	**61.06**	**37.81**	**28.50**	**38.17**	**48.15**	**74.27**	**41.44**	**33.78**
		LSI	**152.35**	**77.22**	**83.33**	**80.00**	**94.87**	**120.53**	**68.33**	**69.44**	**75.38**	**87.18**	**152.35**	**77.22**	**83.33**	**80.00**	**94.87**
	VSM-PMI	VSM	**34.17**	**31.34**	**37.29**	**30.51**	**27.22**	**69.22**	**63.13**	**71.99**	**67.20**	**51.86**	**62.59**	**55.98**	**62.42**	**67.20**	**49.22**
		PMI	**28.87**	**30.02**	**56.02**	**22.37**	**20.09**	**61.29**	**61.48**	**96.59**	**55.19**	**43.31**	**53.97**	**54.39**	**85.43**	**55.19**	**40.82**
	LSI-PMI	LSI	**72.44**	**44.17**	**41.67**	**40.00**	**50.00**	**140.68**	**85.57**	**88.89**	**84.62**	**94.87**	**180.83**	**95.90**	**88.89**	**113.85**	**116.67**
		PMI	-10.12	**19.26**	**55.06**	**2.59**	-2.98	**25.45**	**53.51**	**106.75**	**35.28**	**26.04**	**46.38**	**62.05**	**106.75**	**56.70**	**40.14**
CoreCLR	VSM-LSI	VSM	**38.51**	**49.58**	**80.00**	**70.83**	**54.55**	**38.10**	**52.10**	**100.00**	**75.00**	**48.48**	**38.51**	**49.58**	**80.00**	**70.83**	**54.55**
		LSI	**66.41**	**62.73**	**80.00**	**57.69**	**54.55**	**65.92**	**65.47**	**100.00**	**61.54**	**48.48**	**66.41**	**62.73**	**80.00**	**57.69**	**54.55**
	VSM-PMI	VSM	**79.59**	**65.66**	**70.00**	**95.83**	**106.06**	**166.70**	**153.86**	**200.00**	**220.83**	**181.82**	**190.60**	**163.84**	**180.00**	**200.00**	**206.06**
		PMI	-45.30	-46.95	-46.88	-52.53	-46.88	-18.77	-18.70	-6.25	-22.22	-27.34	-11.48	-15.51	-12.50	-27.27	-21.09
	LSI-PMI	LSI	**103.11**	**98.78**	**110.00**	**80.77**	**69.70**	**202.40**	**207.16**	**290.00**	**184.62**	**172.73**	**335.82**	**288.44**	**360.00**	**265.38**	**266.67**
		PMI	-48.50	-41.49	-34.38	-52.53	-56.25	-23.33	-9.58	**21.88**	-25.25	-29.69	**10.49**	**14.34**	**43.75**	-4.04	-5.47
Angular.js	VSM-LSI	VSM	-20.46	-11.14	-11.54	-17.89	-13.64	-28.26	-11.30	-11.54	-12.63	-15.45	-20.46	-11.14	-11.54	-17.89	-13.64
		LSI	**73.40**	**53.22**	**70.37**	**39.29**	**48.44**	**56.39**	**52.95**	**70.37**	**48.21**	**45.31**	**73.40**	**53.22**	**70.37**	**39.29**	**48.44**
	VSM-PMI	VSM	**13.14**	**10.00**	**7.69**	**8.42**	**10.91**	**28.30**	**23.33**	**28.85**	**21.05**	**17.27**	**30.60**	**24.87**	**30.77**	**20.00**	**17.27**
		PMI	-3.84	-5.69	-11.11	-1.90	**2.52**	**9.04**	**5.75**	**6.35**	**9.52**	**8.40**	**11.00**	**7.07**	**7.94**	**8.57**	**8.40**
	LSI-PMI	LSI	**38.27**	**33.54**	**48.15**	**21.43**	**25.00**	**95.83**	**92.04**	**133.33**	**71.43**	**64.06**	**103.83**	**81.83**	**114.81**	**67.86**	**71.88**
		PMI	-46.09	-33.59	-36.51	-35.24	-32.77	-23.65	-4.50	.00	-8.57	-11.76	-20.53	-9.58	-7.94	-10.48	-7.56

Table 4.9. Comparing the performance of individual IR methods to the λ-optimized method combinations in terms of RR and AP using Wilcoxon Signed Rank Test, and TR_{10} using McNemar's test.

Project	Method	PMI - LSI			PMI - VSM			LSI - VSM		
		MAP	**MRR**	**TR_{10}**	**MAP**	**MRR**	**TR_{10}**	**MAP**	**MRR**	**TR_{10}**
Kubernetes	**VSM**	-	-	-	**-10.887, 0.000**	**-9.938, 0.000**	**0.000**	**-3.707, 0.000**	**-5.502, 0.000**	**0.000**
	LSI	**-13.102, 0.000**	**-11.025, 0.000**	**0.000**	-	-	-	**-11.729, 0.000**	**-9.704, 0.000**	**0.000**
	PMI	**-5.820, 0.000**	**-7.590, 0.000**	**0.000**	**-6.992, 0.000**	**-6.592, 0.000**	**0.000**	-	-	-
CoreCLR	**VSM**	-	-	-	**-11.321, 0.000**	**-11.099, 0.000**	**0.000**	-0.202, 0.840	-1.284, 0.199	**0.001**
	LSI	**-12.109, 0.000**	**-11.803, 0.000**	**0.000**	-	-	-	**-8.330, 0.000**	**-7.966, 0.000**	**0.000**
	PMI	-0.616, 0.538	-0.251, 0.802	0.442	**-3.881, 0.000**	**-3.995, 0.000**	**0.000**	-	-	-
Angular.js	**VSM**	-	-	-	**-6.6864, 0.000**	**-5.261, 0.000**	**0.000**	**-4.959, 0.000**	**-2.562, 0.010**	**0.003**
	LSI	**-9.715, 0.000**	**-8.770, 0.000**	**0.000**	-	-	-	**-9.334, 0.000**	**-7.987, 0.000**	**0.000**
	PMI	**-3.525, 0.000**	-1.600, 0.11	0.137	**-2.097, 0.036**	-1.205, 0.228	**0.002**	-	-	-

References

[1] A. Abadi, M. Nisenson, and Y. Simionovici. A traceability technique for specifications. In *International Conference on Program Comprehension*, pages 103–112, 2008.

[2] S. L. Abebe, S. Haiduc, A. Marcus, P. Tonella, and G. Antoniol. Analyzing the evolution of the source code vocabulary. In *European Conference on Software Maintenance and Reengineering*, pages 189–198, 2009.

[3] E. Agirre, E. Alfonseca, K. Hall, J. Kravalova, M. Paşca, and A. Soroa. A study on similarity and relatedness using distributional and wordnet-based approaches. In *Human Language Technologies: Annual Conference of the North American Chapter of the Association for Computational Linguistics*, pages 19–27, 2009.

[4] N. Anquetil and T. Lethbridge. Assessing the relevance of identifier names in a legacy software system. In *Conference of the Centre for Advanced Studies on Collaborative Research*, pages 4–14, 1998.

[5] G. Antoniol, G. Canfora, G. Casazza, A. De Lucia, and E. Merlo. Recovering traceability links between code and documentation. *IEEE Transactions on Software Engineering*, 28(10):970–983, 2002.

[6] J. Anvik, L. Hiew, and G. Murphy. Coping with an open bug repository. In *Workshop on Eclipse technology eXchange*, pages 35–39, 2005.

[7] T. Ball, J.-M. Kim, A. A. Porter, and H. P. Siy. If your version control system could talk. In *Process Modelling and Empirical Studies of Software Engineering*, 1997.

[8] T. Ball, M. Naik, and S. Rajamani. From symptom to cause: Localizing errors in counterexample traces. In *ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 97–105, 2003.

[9] G. Bavota, R. Oliveto, M. Gethers, D. Poshyvanyk, and A. De Lucia. Methodbook: Recommending move method refactorings via relational topic models. *IEEE Transactions on Software Engineering*, 4(7):671–694, 2014.

[10] M. Beard, N. Kraft, L. Etzkorn, and S. Lukins. Measuring the accuracy of information retrieval based bug localization techniques. In *Working Conference on Reverse Engineering*, pages 124–128, 2011.

[11] N. Bettenburg, S. Just, A. Schröter, C. Weiss, R. Premraj, and T. Zimmermann. What makes a good bug report? In *International Symposium on Foundations of software engineering*, pages 308–318, 2008.

[12] D. Binkley and D. Lawrie. Information retrieval applications in software maintenance and evolution. *Encyclopedia of Software Engineering*, pages 454–463, 2010.

[13] D. Binkley and D. Lawrie. Learning to rank improves ir in se. In *International Conference on Software Maintenance and Evolution*, pages 441–445, 2014.

[14] D. Blei, A. Ng, and M. Jordan. Latent Dirichlet Allocation. *Journal of Machine Learning Research*, 3:993–1022, 2003.

[15] A. Boulanger. Open-source versus proprietary software: Is one more reliable and secure than the other? *IBM Systems Journal*, 44(2):239–248, 2005.

[16] G. Bouma. Normalized (pointwise) mutual information in collocation extraction. In *Proceedings of GSCL*, pages 31–40, 2009.

[17] A. Budanitsky and G. Hirst. Evaluating wordnet-based measures of lexical semantic relatedness. *Computer Linguistics*, 32(1):13–47, 2006.

[18] J. Bullinaria and J. Levy. Extracting semantic representations from word co-occurrence statistics: A computational study. *Behavior Research Methods*, 39(3):510–526, 2007.

[19] J. Cabot, J. L. C. Izquierdo, V. Cosentino, and B. Rolandi. Exploring the use of labels to categorize issues in open-source software projects. In *International Conference on Software Analysis*, pages 550–554, 2015.

[20] J. Chang and D. Blei. Hierarchical relational models for document networks. *The Annals of Applied Statistics*, 4(1):124–150, 2010.

[21] O. Chaparro, J. Lu, F. Zampetti, L. Moreno, M. Di Penta, A. Marcus, G. Bavota, and V. Ng. Detecting missing information in bug descriptions. In *Joint Meeting on Foundations of Software Engineering*, pages 96–407, 2017.

[22] P. Chen and S.-J. Lin. Automatic keyword prediction using google similarity distance. *Expert System With Application*, 37(3):1928–1938, 2010.

[23] K. Church and P. Hanks. Word association norms, mutual information, and lexicography. *Computer Linguistics*, 16(1):22–29, 1990.

[24] P. Ciancarini, F. Poggi, D. Rossi, and A. Sillitti. Analyzing and predicting concurrency bugs in open source systems. In *International Joint Conference on Neural Networks*, pages 721–728, 2017.

[25] R. Cilibrasi and P. Vitanyi. The google similarity distance. *IEEE Transactions on Knowledge and Data Engineering*, 19(3):370–383, 2007.

[26] B. Cleary, C. Exton, J. Buckley, and M. English. An empirical analysis of information retrieval based concept location techniques in software comprehension. *Empirical Software Engineering*, 14(1):93–130, 2009.

[27] M. Codoban, S. Ragavan, D. Dig, and B. Bailey. Software history under the lens: a

study on why and how developers examine it. In *IEEE International Conference on Software Maintenance and Evolution*, pages 1–10, 2015.

[28] T. Cover and J. Thomas. *Elements of Information Theory*. Wiley-Interscience, 1991.

[29] N. Craswell. Mean reciprocal rank. In *Encyclopedia of Database Systems*, pages 1703–1703. Springer, 2009.

[30] K. Crowston and B. Scozzi. Bug fixing practices within free/libre open source software development teams. *Journal of Database Management*, 19(2):1–30, 2008.

[31] V. Dallmeier and T. Zimmermann. Extraction of bug localization benchmarks from history. In *International Conference on Automated Software Engineering*, pages 433–436, 2007.

[32] S. Davies and M. Roper. What's in a bug report? In *International Symposium on Empirical Software Engineering and Measurement*, pages 1–10, 2014.

[33] A. De Lucia, M. Di Penta, R. Oliveto, A. Panichella, and S. Panichella. Improving IR-based traceability recovery using smoothing filters. In *International Conference on Program Comprehension*, pages 21–30, 2011.

[34] A. De Lucia, R. Oliveto, and P. Sgueglia. Incremental approach and user feedbacks: A silver bullet for traceability recovery. In *International Conference on Software Maintenance*, pages 299–309, 2006.

[35] A. Dean and D. Voss. Design and analysis of experiments, 1999.

[36] S. Deerwester, S. Dumais, G. Furnas, T. Landauer, and R. Harshman. Indexing by latent semantic analysis. *Journal of the American Society for Information Science*, 41(6):391–407, 1990.

[37] F. Deiβenböck and M. Pizka. Concise and consistent naming. In *International Workshop on Program Comprehesnion*, pages 97–106, 2005.

[38] L. Deng and D. Yu. Deep learning: methods and applications. *Foundations and trends in signal processing*, 7(3–4):7, 2014.

[39] J. Dinkelacker, P. Garg, R. Miller, and D. Nelson. Progressive open source. In *International Conference on Software Engineering*, pages 177–184, 2002.

[40] S. Dumais, G. Furnas, T. Landauer, S. Deerwester, and R. Harshman. Using latent semantic analysis to improve access to textual information. In *SIGCHI Conference on Human Factors in Computing Systems*, pages 281–285, 1988.

[41] A. L. Edwards. Note on the "correction for continuity" in testing the significance of the difference between correlated proportions. *Psychometrika*, 13(3):185–187, 1948.

[42] B. O. Einar Host. The programmer's lexicon, volume i: The verbs. In *Working Conference on Source Code Analysis and Manipulation*, pages 193–202, 2007.

[43] M. V. Erp and L. Schomaker. Variants of the Borda count method for combining ranked classifier hypotheses. In *International Workshop On Frontiers In Handwriting Recognition*, 2000.

[44] L. Etzkorn, C. Davis, and L. Bowen. The language of comments in computer software: A sublanguage of english. *Journal of Pragmatics*, 33(11):1731–1756, 2001.

[45] M. Feilkas, D. Ratiu, and E. Jurgens. The loss of architectural knowledge during system evolution: An industrial case study. In *International Conference on Program Comprehension*, pages 188–197, 2009.

[46] W. Frakes and B. Nejmeh. Software reuse through information retrieval. *SIGIR Forum*, 21(1):30–36, 1987.

[47] A. Fujii. Enhancing patent retrieval by citation analysis. In *International ACM SIGIR Conference on Research and Development in Information Retrieval*, pages 793–794, 2007.

[48] M. Gabel and S. Zhendong. A study of the uniqueness of source code. In *ACM SIGSOFT International Symposium on Foundations of Software Engineering*, pages 147–156, 2010.

[49] E. Gabrilovich and S. Markovitch. Computing semantic relatedness using wikipedia-based explicit semantic analysis. In *International Joint Conference on Artificial Intelligence*, pages 1606–1611, 2007.

[50] M. Gethers, R. Oliveto, D. Poshyvanyk, and A. De Lucia. On integrating orthogonal information retrieval methods to improve traceability recovery. In *International Conference on Software Maintenance*, pages 133–142, 2011.

[51] J. Gracia, R. Trillo, M. Espinoza, and E. Mena. Querying the web: A multiontology disambiguation method. In *International Conference on Web Engineering*, pages 241–248, 2006.

[52] W. Guo, H. Li, H. Ji, and M. Diab. Linking tweets to news: A framework to enrich short text data in social media. In *Annual Meeting of the Association for Computational Linguistics*, pages 239–249, 2013.

[53] S. Haiduc, J. Aponte, L. Moreno, and A. Marcus. On the use of automated text summarization techniques for summarizing source code. In *Working Conference on Reverse Engineering*, pages 35–44, 2010.

[54] S. Haiduc and A. Marcus. On the use of domain terms in source code. In *IEEE International Conference on Program Comprehension*, pages 113–122, 2008.

[55] Ø. Hauge, C. Ayala, and R. Conradi. Adoption of open source software in software-intensive organizations–a systematic literature review. *Information and Software Technology*, 52(11):1133–1154, 2010.

[56] J. H. Hayes, A. Dekhtyar, and J. Osborne. Improving requirements tracing via information retrieval. In *International Conference on Requirements Engineering*, pages 138–147, 2003.

[57] E. Hill, Z. Fry, H. Boyd, G. Sridhara, Y. Novikova, L. Pollock, and K. Vijay-Shanker. Amap: Automatically mining abbreviation expansions in programs to enhance software maintenance tools. In *Working Conference on Mining Software Repositories*, pages 79–88, 2008.

[58] A. Hindle, E. Barr, Z. Su, M. Gabel, and P. Devanbu. On the naturalness of software. In *International Conference on Software Maintenance*, pages 837–847, 2012.

[59] A. Holzinger, P. Yildirim, M. Geier, and K.-M. Simonic. Quality-based knowledge discovery from medical text on the web. In G. Pasi, G. Bordogna, and L. Jain, editors, *Quality Issues in the Management of Web Information*, pages 145–158. Springer Berlin Heidelberg, 2013.

[60] C. Huntley. Organizational learning in open-source software projects: an analysis of debugging data. *IEEE Transactions on Engineering Management*, 50(4):485–493, 2003.

[61] R. Jacobs. Methods for combining experts' probability assessments. *Methods*, 7(5):867–888, 1995.

[62] M. R. Karim. Key features recommendation to improve bug reporting. In *International Conference on Software and System Processes*, pages 1–4, 2019.

[63] S. Khatiwada, M. Kelly, and A. Mahmoud. STAC: A tool for static textual analysis of code. In *International Conference on Program Comprehension*, pages 1–3, 2016.

[64] S. Khatiwada, M. Tushev, and A. Mahmoud. Just enough semantics: An information theoretic approach for ir-based software bug localization. *Information and Software Technology*, 93:45–57, 2017.

[65] P. Kochhar, Y. Tian, and D. Lo. Potential biases in bug localization: Do they matter? In *international conference on Automated software engineering*, pages 803–814, 2014.

[66] S. Kullback and R. Leibler. On information and sufficiency. *The Annals of Mathematical Statistics*, 22(1):79–86, 1951.

[67] A. N. Lam, A. T. Nguyen, H. A. Nguyen, and T. N. Nguyen. Bug localization with combination of deep learning and information retrieval. In *International Conference on Program Comprehension*, pages 218–229, 2017.

[68] T. LaToza and B. Myers. Hard-to-answer questions about code. In *Evaluation and Usability of Programming Languages and Tools*, page 8, 2010.

[69] D. Lawrie, C. Morrell, H. Feild, and D. Binkley. What's in a name? a study of identifiers. In *International Conference on Program Comprehension*, pages 3–12, 2006.

[70] T. Le, R. Oentaryo, and D. Lo. Information retrieval and spectrum based bug localization: Better together. In *Joint Meeting on Foundations of Software Engineering*, pages 579–590, 2015.

[71] T.-D. Le, F. Thung, and D. Lo. Predicting effectiveness of IR-based bug localization techniques. In *International Symposium on Software Reliability Engineering*, pages 335–345, 2014.

[72] D. Lin. An information-theoretic definition of similarity. In *International Conference on Machine Learning*, pages 296–304, 1998.

[73] M. Linares-Vásquez, K. Hossen, H. Dang, H. Kagdi, M. Gethers, and D. Poshyvanyk. Triaging incoming change requests: Bug or commit history, or code authorship? In *International Conference on Software Maintenance*, pages 451–460, 2012.

[74] C. Liu, X. Yan, L. Fei, J. Han, and S. Midkiff. Sober: Statistical model-based bug localization. In *European Software Engineering Conference*, pages 286–295, 2005.

[75] S. Lohar, S. Amornborvornwong, A. Zisman, and J. Cleland-Huang. Improving trace accuracy through data-driven configuration and composition of tracing features. In *Joint Meeting on Foundations of Software Engineering*, pages 378–388, 2013.

[76] S. Lukins, N. Kraft, and L. Etzkorn. Source code retrieval for bug localization using latent dirichlet allocation. In *Reverse Engineering*, pages 155–164, 2008.

[77] Y. Maarek, D. Berry, and G. Kaiser. An information retrieval approach for automatically constructing software libraries. *IEEE Transactions on Software Engineering*, 17(8):800–813, 1991.

[78] A. Mahmoud and G. Bradshaw. Estimating semantic relatedness in source code. *ACM Transactions on Software Engineering and Methodology*, 25(1):1–35, 2015.

[79] A. Mahmoud and N. Niu. Supporting requirements traceability through refactoring. In *International Requirements Engineering Conference*, pages 32–41, 2013.

[80] A. Mahmoud and N. Niu. On the role of semantics in automated requirements tracing. *Requirements Engineering*, 20(3):281–300, 2015.

[81] C. Manning, P. Raghavan, H. Schütze, et al. *Introduction to information retrieval*. Cambridge university press Cambridge, 2008.

[82] C. Manning and H. Schütze. *Foundations of Statistical Natural Language Processing*. MIT Press, 1999.

[83] A. Marcus and J. Maletic. Recovering documentation-to-source-code traceability links using latent semantic indexing. In *International Conference on Software Engineering*, pages 125–135, 2003.

[84] A. Marcus, V. Rajlich, J. Buchta, M. Petrenko, and A. Petrenko. Static techniques for concept location in object-oriented code. In *Program Comprehension*, pages 33–42, 2005.

[85] A. Marcus, A. Sergeyev, V. Rajlich, and J. Maletic. An information retrieval approach to concept location in source code. In *Working Conference on Reverse Engineering*, pages 214–223, 2004.

[86] R. Mihalcea, C. Corley, and C. Strapparava. Corpus-based and knowledge-based measures of text semantic similarity. In *National Conference on Artificial Intelligence*, pages 775–780, 2006.

[87] C. Mills, J. Pantiuchina, E. Parra, G. Bavota, and S. Haiduc. Are bug reports enough for text retrieval-based bug localization? In *International Conference on Software Maintenance and Evolution*, pages 381–392, 2018.

[88] L. Moreno, J. Treadway, A. Marcus, and W. Shen. On the use of stack traces to improve text retrieval-based bug localization. In *International Conference on Software Maintenance and Evolution*, pages 151–160, 2014.

[89] D. Newman, Y. Noh, E. Talley, S. Karimi, and T. Baldwin. Evaluating topic models for digital libraries. In *Joint Conference on Digital Libraries*, pages 215–224, 2010.

[90] A. Nguyen, T. Nguyen, J. Al-Kofahi, H. Nguyen, and T. Nguyen. A topic-based approach for narrowing the search space of buggy files from a bug report. In *Automated Software Engineering*, pages 263–272, 2011.

[91] R. Oliveto, M. Gethers, D. Poshyvanyk, and A. De Lucia. On the equivalence of information retrieval methods for automated traceability link recovery. In *International Conference on Program Comprehension*, pages 68–71, 2010.

[92] T. Ostrand, E. Weyuker, and R. Bell. Predicting the location and number of faults in large software systems. *IEEE Transactions on Software Engineering*, 31(4):340–355, 2005.

[93] C. Parnin and A. Orso. Are automated debugging techniques actually helping programmers? In *International Symposium on Software Testing and Analysis*, pages 199–209, 2011.

[94] S. Polisetty, A. Miranskyy, and A. Başar. On usefulness of the deep-learning-based bug

localization models to practitioners. In *International Conference on Predictive Models and Data Analytics in Software Engineering*, pages 16–25, 2019.

[95] L. Pollock, K. Vijay-Shanker, E. Hill, G. Sridhara, and D. Shepherd. Natural language-based software analyses and tools for software maintenance. In *Software Engineering*, pages 94–125. Springer, 2013.

[96] J. Ponte and W. Croft. A language modeling approach to information retrieval. In *Annual International ACM SIGIR Conference on Research and Development in Information Retrieval*, pages 275–281, 1998.

[97] F. Porter. *An algorithm for suffix stripping*, pages 313–316. Morgan Kaufmann Publishers Inc., 1997.

[98] D. Poshyvanyk, Y. Gueheneuc, A. Marcus, G. Antoniol, and V. Rajlich. Feature location using probabilistic ranking of methods based on execution scenarios and information retrieval. *Software Engineering, IEEE Transactions*, 33(6):420–432, 2007.

[99] V. Rajlich and N. Wilde. The role of concepts in program comprehension. In *International Workshop on Program Comprehension*, pages 271–278, 2007.

[100] S. Rao and A. Kak. Retrieval from software libraries for bug localization: A comparative study of generic and composite text models. In *Working Conference on Mining Software Repositories*, pages 43–52, 2011.

[101] S. Rao and A. Kak. morebugs: A new dataset for benchmarking algorithms for information retrieval from software repositories. Technical report, Purdue University, School of Electrical and Computer Engineering, 2013.

[102] M. Renieres and S. P. Reiss. Fault localization with nearest neighbor queries. In *Automated Software Engineering*, pages 30–39, 2003.

[103] S. Robertson, S. Walker, and M. Beaulieu. Experimentation as a way of life: Okapi at trec. *Information Process Management*, 36(1):95–108, 2000.

[104] R. Saha, M. Lease, S. Khurshid, and D. Perry. Improving bug localization using structured information retrieval. In *Automated Software Engineering*, pages 345–355, 2013.

[105] G. Salton, A. Wong, and C. Yang. A vector space model for automatic indexing. *Communications of ACM*, 18(11):613–620, 1975.

[106] D. Silva and M. Tulio Valente. Refdiff: Detecting refactorings in version histories. In *International Conference on Mining Software Repositories*, pages 269–279, 2017.

[107] B. Sisman and A. Kak. Incorporating version histories in information retrieval based bug localization. In *Conference on Mining Software Repositories*, pages 50–59, 2012.

[108] E. Soloway and K. Ehrlich. Empirical studies of programming knowledge. *IEEE Transactions on Software Engineering*, 10(5):595–609, 1984.

[109] D. Sousa, L. Sarmento, and E. Rodrigues. Characterization of the twitter replies network: Are user ties social or topical? In *International Workshop on Search and Mining User-generated Contents*, pages 63–70, 2010.

[110] M. Strube and S. Ponzetto. Wikirelate! computing semantic relatedness using wikipedia. In *National Conference on Artificial Intelligence*, pages 1419–1424, 2006.

[111] R. Tairas and J. Gray. An information retrieval process to aid in the analysis of code clones. *Empirical Software Engineering*, 14(1):33–56, 2009.

[112] C. Tantithamthavorn, A. Ihara, and K. ichi Matsumoto. Using co-change histories to improve bug localization performance. In *International Conference on Software Engineering, Artificial Intelligence, Networking and Parallel/Distributed Computing*, pages 543–548, 2013.

[113] S. Teufel. An overview of evaluation methods in trec ad hoc information retrieval and trec question answering. In *Evaluation of text and speech systems*, pages 163–186, 2007.

[114] S. Thomas. Mining software repositories using topic models. In *International Conference on Software Engineering*, pages 1138–1139, 2011.

[115] S. Thomas, M. Nagappan, D. Blostein, and A. Hassan. The impact of classifier configuration and classifier combination on bug localization. *IEEE Transactions on Software Engineering*, 39(10):1427–1443, 2013.

[116] F. Thung, T. Le, P. Kochhar, and D. Lo. Buglocalizer: Integrated tool support for bug localization. In *ACM SIGSOFT International Symposium on Foundations of Software Engineering*, pages 767–770, 2014.

[117] P. Turney. Mining the web for synonyms: PMI-IR versus LSA on TOEFL. In *European Conference on Machine Learning*, pages 491–502, 2001.

[118] H. Valdivia-Garcia, E. Shihab, and M. Nagappan. Characterizing and predicting blocking bugs in open source projects. In *Journal of Systems and Software*, pages 44–58, 2016.

[119] H. Wang and H. Kagdi. A conceptual replication study on bugs that get fixed in open source software. In *International Conference on Software Maintenance and Evolution*, pages 299–310, 2018.

[120] S. Wang, F. Khomh, and Y. Zou. Improving bug localization using correlations in crash reports. In *Working Conference on Mining Software Repositories*, pages 247–256, 2013.

[121] S. Wang and D. Lo. Compositional vector space models for improved bug localization.

In *International Conference on Software Maintenance and Evolution*, pages 171–180, 2014.

[122] S. Wang and D. Lo. Version history, similar report, and structure: Putting them together for improved bug localization. In *International Conference on Program Comprehension*, pages 53–63, 2014.

[123] S. Wang and D. Lo. Amalgam+: Composing rich information sources for accurate bug localization. *Journal of Software: Evolution and Process*, 28(10):921–942, 2016.

[124] R. L. Winkler and R. T. Clemen. Multiple experts vs. multiple methods: Combining correlation assessments. *Decision Analysis*, 1(3):167–176, 2004.

[125] C. Wohlin, P. Runeson, M. Höst, M. Ohlsson, B. Regnell, and A. Wesslén. *Experimentation in Software Engineering*. Springer, 2012.

[126] W. Woon and S. Madnick. Asymmetric information distances for automated taxonomy construction. *Knowledge and Information Systems*, 21(1):91–111, 2009.

[127] R. Wu, M. Wen, S.-C. Cheung, and H. Zhang. Changelocator: locate crash-inducing changes based on crash reports. *Empirical Software Engineering*, 23(5):2866–2900, 2018.

[128] Z. Xiang, K. Wöber, and D. Fesenmaier. Representation of the online tourism domain in search engines. *Journal of travel research*, 47(2):137–150, 2008.

[129] Y. Xiao, J. Keung, K. E. Bennin, and Q. Mi. Machine translation-based bug localization technique for bridging lexical gap. *Information and Software Technology*, 99(10):58–61, 2018.

[130] X. Ye, R. Bunescu, and C. Liu. Learning to rank relevant files for bug reports using domain knowledge. In *ACM SIGSOFT International Symposium on Foundations of Software Engineering*, pages 689–699, 2014.

[131] Y. Ye and K. Kishida. Toward an understanding of the motivation open source software developers. In *International Conference on Software Engineering*, pages 419–429, 2003.

[132] K. C. Youm, J. Ahn, J. Kim, and E. Lee. Bug localization based on code change histories and bug reports. In *Asia-Pacific Software Engineering Conference*, pages 190–197, 2015.

[133] M. S. Zanetti, I. Scholtes, C. J. Tessone, and F. Schweitzer. Categorizing bugs with social networks: a case study on four open source software communities. In *International Conference on Software Engineering*, pages 1032–1041, 2013.

[134] A. Zeller and R. Hildebrandt. Simplifying and isolating failure-inducing input. *IEEE Transactions on Software Engineering*, 28(2):183–200, 2002.

[135] H. Zhang. An investigation of the relationships between lines of code and defects. In *International Conference on Software Maintenance*, pages 274–283, 2009.

[136] X. Zhang, H. He, N. Gupta, and R. Gupta. Experimental evaluation of using dynamic slices for fault location. In *International Symposium on Automated Analysis-driven Debugging*, pages 33–42, 2005.

[137] J. Zhou, H. Zhang, and D. Lo. Where should the bugs be fixed? More accurate information retrieval-based bug localization based on bug reports. In *International Conference on Software Engineering*, pages 14–24, 2012.

[138] T. Zimmermann, S. Kim, A. Zeller, and J. Whitehead. Mining version archives for co-changed lines. In *Mining Software Repositories*, volume 6, pages 72–75, 2006.

Vita

Saket Khatiwada, born in Biratnagar, Nepal, received his B.S. degree in Computer Science, with a minor in Mathematics and Physics, from the Southeastern Louisiana University in 2014. During his undergraduate, he worked as a Web Developer for the university and worked as a Senior Developer after graduation. This experience, among others, led him to pursue a graduate degree in the Department of Computer Science and Engineering at Louisiana State University. He worked under the mentorship of Dr. Anas Mahmoud and published several papers in multiple prestigious venues. His main research interests include bug localization, software debugging, and information retrieval.