On Combining IR Methods to Improve Bug Localization

Saket Khatiwada, Miroslav Tushev, and Anas Mahmoud
The Division of Computer Science and Engineering
Louisiana State University
skhati1,mtushe1,amahmo4@lsu.edu

ABSTRACT

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 has to 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 paper, 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 be used to 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 as well as 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.

CCS CONCEPTS

• **Software and its engineering → Maintaining software**.

KEYWORDS

Information Retrieval, Bug Localization, Software Debugging

ACM Reference Format:
Saket Khatiwada, Miroslav Tushev, and Anas Mahmoud. 2020. On Combining IR Methods to Improve Bug Localization. In *28th International Conference on Program Comprehension (ICPC '20), October 5–6, 2020, Seoul, Republic of Korea.* ACM, New York, NY, USA, 11 pages. https://doi.org/10.1145/3387904.3389280

1 INTRODUCTION

Despite the substantial resources being devoted to deliver high quality software, software systems are still shipped with defects. Whenever a bug is reported, developers go through their code to find any code fragments related to the bug, a process that is known as *bug localization*. However, when software systems get large, localizing bugs manually can become a tedious and error-prone task [34, 43]. This problem can be particularly challenging in open source environments, where projects can have multiple active branches that are maintained by distributed developer teams [48]. To minimize this effort, contemporary bug localization tools employ conventional Information Retrieval (IR) methods for automated support. The main objective is to help developers narrow down their search space when looking for buggy code artifacts [29, 38]. Generally speaking, IR methods match the textual description of bugs, often expressed in natural language, with source code. The underlying assumption is that code fragments that share some sort of textual similarity with a bug's description are likely to be related to the bug [24, 37, 38].

IR-based bug localization methods are appealing for their low computational cost and the fact that they can be independent of the programming language [3, 21, 24, 27, 40, 54]. 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 in order to locate relevant code (true positives) and to 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 [4, 15, 16, 45, 49]. 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) as well as help to retrieve links that are missed by individual methods (recall). However, existing hybrid methods either treat individual IR methods equally or rely on existing heuristics for assigning confidence levels to each individual IR method in the hybrid combination [16, 45]. These tactics can limit the improvement in the performance as different IR methods perform differently, thus, 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 [2, 7, 16, 20, 24, 30, 40, 45, 52, 54]. Specifically, our set of IR methods consists of: Vector Space Model (VSM) [41] as a representative of string matching methods, Latent Semantic Indexing (LSI) [11] as a representative of semantically-enabled methods, Jensen-Shannon Model (JSM) [1] as a representative of probabilistic methods, and Pointwise Mutual Information (PMI) [8] as a representative of information-theoretic text retrieval methods.

ICPC '20, October 5–6, 2020, Seoul, Republic of Korea

ACM ISBN 978-1-4503-7958-8/20/05…$15.00
https://doi.org/10.1145/3387904.3389280

Our analysis in this paper is conducted using five benchmark systems obtained from a broad range of application domains. Our objectives are to ***a)*** determine the nature of 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.

The rest of this paper is organized as follows. Section 2 provides a brief discussion of the different IR methods and hybrid approaches investigated in our analysis. Section 3 describes our experimental setup. Section 4 presents our results. Section 5 discusses our main findings. Section 6 reviews seminal related work and positions our work within existing literature. Section 7 discusses the potential limitations of our study. Finally, Section 8 concludes the paper and describes several directions of future work.

2 APPROACH AND RESEARCH QUESTIONS

In this section, we introduce the various IR methods used in our analysis, describe the hybrid approaches used for combining these methods, and present our main research questions.

2.1 Information Retrieval Methods

A large number of IR methods have been employed to provide automatic support for developers during bug localization sessions [2, 7, 16, 20, 24, 30, 40, 45, 52, 54]. In our analysis, we select four of these methods that, ***a)*** have been consistently reported to achieve competitive performance, and ***b)*** support relatively different similarity functions. The following is a description of these methods.

2.1.1 Vector Space Model. VSM is an algebraic model that represents a text artifact as an unordered vector of words (also known as a *bag-of-words*) [41]. Using VSM, a corpus (collection of text documents) is converted into a term-document matrix. Each term or word in the matrix represents a single term or word found in the corpus and each column represents an individual document. Each entry in the matrix $w_{i,j}$ carries a specific weight for the term j in the document i. The weight of a term is typically measured using its frequency - inverse document frequency (TF.IDF) [26]. TF.IDF is the product of the number of occurrence of the term in the document (TF) and the term's scarcity across all the documents (IDF). Formally, TF.IDF can be computed as:

$$TF.IDF = f(w,d) \times \log \frac{|D|}{|d_i : w \in d_i \wedge d_i \in D|} \tag{1}$$

where *f(w,d)* is the frequency of occurrences of the term w in the document d, D is the total number of documents in the corpus, and $w \in d$ indicates the presence of the term w in the document d. Similarity between two documents can be determined by calculating the distance between their document vectors. Cosine distance is widely used to measure the similarity between vectors.

2.1.2 Latent Semantic Indexing. LSI [11] is an IR method that is used to identify the latent relationships between terms and concepts contained in a text corpus. LSI draws on the assumption that there is some underlying semantic structure that is partially concealed by the variability of the contextual usage of words in a certain corpus [12]. Formally, LSI starts by constructing a single term-document matrix of size $m \times n$ for the corpus, 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 this matrix to a product of three matrices, expressing the term-document matrix in the form:

$$A = U\Sigma V^T \tag{2}$$

where A is the term-document matrix, U and V are an $m \times r$ and an $r \times n$ orthogonal matrices respectively, and Σ is an $r \times r$ diagonal matrix of the singular values of A. The key objective of LSI is to reduce the dimensionality of the information retrieval problem. This is achieved by deleting all but the k largest values on the diagonal of Σ and the 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 a k-dimensional vector space. The vectors in this space represent the documents in the corpus. The vector representation for the query in this k-dimensional space can be calculated in the same manner as:

$$v = q^T U_k \Sigma_k^{-1} \tag{3}$$

where q is the *TF*-vector for the query. The similarity between the query and a document in the collection can then be computed by calculating the cosine distance between the query computed vector v and the column vector representing the document in the V_k^T matrix.

2.1.3 Jensen-Shannon Model. Presented by Abadi et al. [1], JSM is a probabilistic IR technique that represents each document d in the corpus as a probabilistic distribution over its words. The probability of each word in this distribution is calculated as:

$$p_i = \frac{f(w_i, d)}{T_d} \tag{4}$$

where $f(w_i, d)$ is the frequency of the word w_i in the document d, and T_d is the total number of words in d. The similarity between any two documents (q, d) can be estimated using the distance between their probabilistic distributions $\hat{p_q}$ and $\hat{p_q}$, calculated using the Jensen-Shannon Divergence [9] as:

$$JSM(d,q) = 1 - \left[H(\frac{\hat{p_d} + \hat{p_q}}{2}) - \frac{H(\hat{p_d}) + H(\hat{p_q})}{2} \right] \tag{5}$$

where H is the entropy of the distribution $\hat{p}$, given by:

$$H(\hat{p}) = \sum_{j=1}^{n} \hat{p}(x_j) \cdot \log_2 \hat{p}(x_j) \tag{6}$$

where n is the length of $\hat{p}$. JSM value can be in the range [0, 1], where 1 indicates a perfect similarity. JSM has been used in software traceability tasks and has been reported to outperform methods such as VSM and LSI [1, 31].

2.1.4 Pointwise Mutual Information. PMI is an information-theoretic measure of information overlap, or statistical dependence, between two words. PMI was introduced by Church and Hanks [8], and later used by Turney [46] to identify synonym pairs using Web search results. Formally, the PMI between two words w_1 and w_2 can be measured as the probability of them occurring in the same text

versus the probabilities of them occurring separately. Assuming the corpus contains N artifacts, PMI can be calculated as:

$$PMI = \log_2(\frac{\frac{C(w_1,w_2)}{N}}{\frac{C(w_1)}{N}\frac{C(w_2)}{N}}) = \log_2(\frac{P(w_1,w_2)}{P(w_1)P(w_2)}) \quad (7)$$

where $C(w_1, w_2)$ is the number of documents in the corpus containing both w_1 and w_2, and $C(w_1)$ and $C(w_2)$ are the number of documents containing w_1 and w_2 respectively. If the words w_1 and w_2 are frequently associated, the probability of observing w_1 and w_2 together will be 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 the amount of information acquired about w_1 when observing w_2. The value of PMI can go from $-\infty$ to $+\infty$, where $-\infty$ indicates that the two words are not related, or do not appear together in any of the corpus documents, and $+\infty$ implies a complete co-occurrence between the words. The value of PMI can be normalized to fit in the range [-1, 1] as follows [5]:

$$nPMI = \frac{PMI}{-\log_2(P(w_1,w_2))} = \frac{\log_2(P(w_1)P(w_2))}{\log_2(P(w_1,w_2))} - 1 \quad (8)$$

To utilize PMI in text retrieval, a method such as Text Semantic Similarity (TSS) can be used [28]. TSS is a text based and corpus based measure that estimates semantic similarity between two texts using the pairwise semantic similarity of their individual words. TSS models the semantic similarity of texts as a function of the semantic similarity of their words. In particular, TSS combines information from word-to-word similarity and word specificity to calculate the similarity between texts. Formally, the semantic similarity between two texts T_1 and T_2 can be described as follows:

$$TSS = \frac{1}{2}(\frac{\sum_{i=1}^{|T_1|}(maxSim(w_i,T_2)\times IDF(w_i))}{\sum IDF(w_i)} + \frac{\sum_{j=1}^{|T_2|}(maxSim(w_j,T_1)\times IDF(w_j))}{\sum IDF(w_j)}) \quad (9)$$

where $maxSim(w_i, T_2)$ is the similarity score between w_i from T_1 and its most similar word in T_2. Similarly, $maxSim(w_j, T_1)$ is 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 in the corpus divided by the number of artifacts that contain the word w.

2.2 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 arrives at the conclusion, or the similarity judgment, between a bug report and code artifacts differently. For instance, VSM uses simple word matching to calculate similarity. LSI, on the other hand, calculates similarity between texts by comparing their underlying latent semantic structures (or topics). The hybrid approach assumes that each individual 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 forecast that is more certain than that of each individual expert [18, 50]. This approach has been successfully employed in tasks such as feature location [37] and traceability link recovery [16]. In our analysis, we consider two types of hybrid approaches: optimized and unoptimized.

2.2.1 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 [16]. 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:

$$sim_{i,j}(x,y) = \lambda \times sim_i(x,y) + (1-\lambda)\times sim_j(x,y) \quad (10)$$

Eq. 10 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:

$$sim_i(x,y)_{normalized} = \frac{sim_i(x,y) - \overline{sim_i(X,Y)}}{\sigma(sim_i(X,Y))} \quad (11)$$

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

2.2.2 Unoptimized Hybrid Methods. This set of hybrid methods do not apply any sort of 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 [45]. These methods can be described as follows:

- **Borda Count**: Borda Count [14] is a rank-only combination approach which 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:

$$Borda(k) = \sum_{i=0}^{|C|} M_i - r_{i,k} \quad (12)$$

  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.
- **Score Addition**: Score Addition is a score-based combination approach that sums up the scores assigned by each individual IR methods 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 calculated as:

$$ScoreAddition(k) = \sum_{i=0}^{|C|} s_{i,k} \quad (13)$$

Table 1: The experimental systems used in our analysis.

System	Bug Reports	Source File
AspectJ [39]	318	6503
Eclipse [54]	3075	12300
JodaTime [39]	43	315
SWT [54]	98	484
ZXing [54]	20	391

2.3 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 in Eq.10?
- **RQ_2:** How effective is the hybrid approach in comparison to the individual IR methods?
- **RQ_3:** How effective are the λ-optimized IR-hybrids in comparison to the unoptimized hybrids?
- **RQ_4:** How does the performance of individual IR methods affect the performance of their hybrid pairs?

3 EXPERIMENTAL SETUP

In this section, we describe our empirical investigation, including the systems used in our analysis and the different performance measures used to assess the accuracy of the proposed methods.

3.1 Experimental Systems

Five benchmark open source systems from two datasets are used in our analysis. These systems, described in Table 1, were obtained from the following sources:

- Zhou et al. [54]: This dataset includes bug reports from three popular open source projects, *SWT*, *ZXing*, and *Eclipse*. Each bug report in the dataset consists of the bug's title, its description, and the list of files modified to fix the bug. This dataset consists 98, 20, and 3075 bug reports for *SWT*, *ZXing*, and *Eclipse* respectively.
- moreBugs [39]: This dataset includes bug reports from the *AspectJ* and the *JodaTime* repositories. For each bug report, the dataset 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 *AspectJ* and 43 bug reports for *JodaTime*.

3.2 Evaluation Measures

To evaluate the performance of the proposed methods, we use two conventional IR evaluation measures that are commonly used in the bug localization literature [17, 25, 40]. These measures can be described as follows:

- **Mean Reciprocal Rank (MRR)**: Reciprocal rank (RR) is the multiplicative inverse of the rank of the first correct item of a query. 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 item occurs at rank n, then the reciprocal rank is computed as $\frac{1}{n}$ [10]. MRR is the average of the reciprocal ranks across all queries Q. This measure can be calculated as follows:

$$MRR = \frac{1}{|Q|}\sum_{i=1}^{|Q|}\frac{1}{rank_i} \tag{14}$$

  where $rank_i$ is the rank of the first relevant item 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 item in the first N items of the retrieved list. Different values of N can be used, in our analysis we use $N = \{1, 5, 10\}$. For example, a TR_1= 50% indicates that 50% of the queries returned a correct item as the top ranked item in the list (first hit).

MRR and TR_N are effective indicators of the practicality of IR-based bug localization tools. Specifically, there are typically 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 [40]. In fact, 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 [2, 22, 33].

3.3 Implementation

We start our experimental analysis by indexing source code artifacts in our systems. Indexing is the process of extracting the textual component of code (code lexicon) embedded in identifier names and internal code comments. In our analysis, a code file is treated as a text file. The textual content of each file is extracted using string manipulation [19]. Extracted code identifiers are split into their constituent words using standard camel-casing (e.g., `libraryEntry` is split to `library` and `entry`). During this process, programming reserved words and English stop-words are filtered out (e.g., *int*, *the*). These words are highly unlikely to provide any discriminative information to the retrieval method. The stop-word list provided by StanfordNLP is used in our analysis. The remaining terms are stemmed to their morphological roots using Porter stemmer [36]. The outcome of the indexing process is a compact content descriptor (unordered vector of terms) for each of our code artifacts. The code in our experiment is indexed at a class granularity level (i.e., a code artifact is basically a single class). In particular, since we are mainly dealing with Object Oriented code, we assume that each file holds a single class. Inner classes are considered to be a part of the main class.

The Bluebit Matrix Calculator, a high performance matrix algebra for .NET programming, is used to implement LSI. We adopted a brute-force search strategy to determine the optimal value of k for each query [20]. Specifically, for each query in each of our systems, we generated the LSI space for all k values in the set [50, 100, 150, 200, ..., 900]. The performance in terms of reciprocal rank (RR) was then measured for each query at each k value. The k value which

produced the highest RR was used. In *AspectJ*, *SWT*, and *ZXing*, the majority of queries achieved their highest RR values at k=50. In *Eclipse*, k=100 achieved the highest RR for the majority of the queries. In *JodaTime*, the best RR was achieved at k=300.

4 RESULTS AND ANALYSIS

In this section, we examine the performance of the various IR methods proposed earlier, analyze and compare the performance of the hybrid methods, and describe the λ optimization process.

4.1 Individual Methods Performance

We start our analysis by examining the performance of individual IR methods. Specifically, each method is used to retrieve the code artifacts relevant to each bug report in each of our benchmark systems. Table 2 shows the TR_1, TR_5, TR_{10}, and MRR values for each IR method averaged over each system.

The results show that, on average, PMI outperforms VSM, JSM, and LSI, in terms of MRR, TR_5, and TR_{10}. In general, PMI works better in larger systems, while JSM seems to be working better for smaller systems. This can be explained based on the fact that PMI favors 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 PMI a clear advantage over other IR methods as larger source code files tend to be more defect-prone. Zhang et al. [53] reported that only a small number of largest source files in software systems accounts for a large proportion of the defects. Similar observations were made by Ostrand et al [32] who reported that 20% of the largest files contained 70% of bugs.

We applied Wilcoxon rank sum test to examine the differences in the performance of individual IR methods in terms of MRR. Specifically, we compared the RR values for each query in each system achieved by PMI against the RR values achieved by other methods (VSM, LSI, and JSM). Statistical significance was measured at $p \leq 0.05$. Table 3 report the results of our statistical analysis. The results show that PMI significantly outperformed other methods in three out of five systems used in our analysis. More specifically, in Eclipse, AspectJ, and SWT, PMI managed to achieve statistically significant improvement over other methods. In JodaTime, JSM managed to achieve the best results. However, the improvement over PMI was insignificant. A similar behavior was observed in ZXing; other methods outperformed PMI, with JSM achieving the best results. However, the differences in MRR were insignificant.

4.2 Optimized Hybrids *vs.* Individual Methods

A main question when combining two IR methods using the optimized approach is to determine the λ value that can maximize the retrieval accuracy of the hybrid pair. In the literature, such 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. 10 [16, 18]. 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 individual 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., $\lambda = 0, 0.1, 0.2, 0.3, ..., 1$).

Table 2: The performance of the individual IR methods in terms of TR_1, TR_5, TR_{10}, and MRR

System	Method	TR_1(%)	TR_5(%)	TR_{10}(%)	MRR
AspectJ	VSM	5.66	16.04	22.64	0.12
	LSI	7.55	14.78	22.01	0.12
	JSM	7.55	18.55	23.90	0.13
	PMI	15.72	35.22	45.28	0.25
Eclipse	VSM	8.85	21.53	29.27	0.16
	LSI	18.34	33.92	42.47	0.26
	JSM	14.24	29.98	38.76	0.22
	PMI	17.59	40.36	51.67	0.29
JodaTime	VSM	20.93	51.16	67.44	0.35
	LSI	37.21	53.49	62.79	0.45
	JSM	37.21	58.14	72.09	0.47
	PMI	18.60	62.79	83.72	0.40
SWT	VSM	11.22	34.69	46.94	0.23
	LSI	8.16	19.39	24.49	0.14
	JSM	11.22	29.59	43.88	0.22
	PMI	27.55	68.37	81.63	0.44
ZXing	VSM	30.00	40.00	55.00	0.37
	LSI	30.00	45.00	45.00	0.38
	JSM	35.00	55.00	65.00	0.44
	PMI	25.00	45.00	55.00	0.34

Table 3: Comparing methods performance against PMI in terms of MRR using Wilcoxon Rank Sum Test

	PMI-VMS	PMI-LSI	PMI-JSM
AspectJ	p < 0.00, Z =-9.89	p < 0.00, Z=-8.41	p < 0.00, Z=-9.08
Eclipse	p < 0.00, Z=-26.55	p < 0.00, Z=-7.08	p < 0.00, Z=-15.28
JodaTime	p = 0.39, Z=-0.87	p = 0.75, Z=-0.32	p = 0.54, Z=-0.611
SWT	p < 0.00, Z=-4.64	p < 0.00, Z=-6.15	p < 0.00, Z=-5.08
Zxing	p = 0.96, Z=-0.05	p = 0.88, Z=-0.15	p = 0.27, Z=-1.108

Values of λ that maximized the performance for each hybrid pair of methods over each system were then averaged. For example, Fig. 1 shows the performance (MRR) of VSM 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 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 is shown in Table 4. For instance, 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 a lower confidence (λ = 0.2) while PMI is assigned a higher confidence (1 - λ = 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. In particular, PMI achieved the best performance individually, followed by JSM, LSI and finally VSM. These results answers our first research question (**RQ_1**). In general, 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.

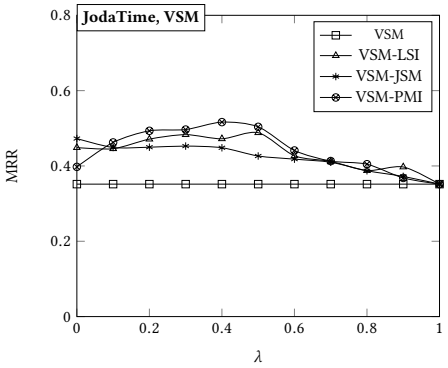

Figure 1: Optimizing λ for VSM in JodaTime.

Table 4: Average near-optimal values of λ of the hybrid methods across all experimental systems (Optimized for best MRR).

	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	-

To answer our second research question (**RQ$_2$**), we generate the set of ranked lists for each system using all combinations of the IR methods given the λ values identified earlier. The results are shown in Table 5. The table shows the percentage change (potential loss or gain) in MRR. Specifically, assuming 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:

$$\frac{P_{ij} - P_i}{P_i} \times 100\% \tag{15}$$

For example, in Table 5, 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 method i in Eq. 10). The row in the table shows that the MRR for the VSM-LSI pair increased by 20.17% when compared to the MRR achieved by VSM alone. To detect statistical significance in the loss/gain results, we used Wilcoxon signed-rank test. The results in Table 6 and Table 7 show that, except for 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 **RQ$_2$**.

4.3 Optimized vs. Unoptimized Methods

To answer our third research question (**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 the ranking of true positives between the different methods. The results of our analysis are shown in Table 8. On average,

Table 5: The performance gain (%MRR) of the hybrid methods in comparison to the individual IR methods.

Method	System	VSM	LSI	JSM	PMI
VSM	AspectJ	-	20.17	27.04	104.80
	Eclipse	-	58.35	35.32	85.74
	Joda	-	38.83	28.72	40.27
	SWT	-	-3.47	17.32	114.53
	ZXing	-	28.56	19.88	-1.70
Average		-	28.49	25.66	68.737
LSI	AspectJ	13.46	-	16.67	115.99
	Eclipse	-6.42	-	3.29	38.09
	Joda	8.79	-	25.06	31.70
	SWT	56.79	-	43.66	212.39
	ZXing	24.21	-	19.53	26.07
Average		19.36	-	21.64	84.85
JSM	AspectJ	13.75	10.69	-	74.55
	Eclipse	-5.58	21.96	-	37.24
	Joda	-4.23	18.74	-	28.18
	SWT	27.96	-3.53	-	109.01
	ZXing	1.08	4.31	-	-6.11
Average		6.60	10.43	-	48.58
PMI	AspectJ	-6.75	4.91	-11.24	-
	Eclipse	0.69	26.71	6.68	-
	Joda	24.09	48.69	52.42	-
	SWT	15.28	3.35	2.97	-
	ZXing	5.96	40.66	20.04	-
Average		7.85	24.86	14.18	-

the optimized approach managed to outperform the other unoptimized methods that treat all individual IR methods equally. To test for statistical significance, we used the McNemar's test [13]. This test uses a categorical explanatory variable to show if paired observations in two groups differ significantly in terms of the dependent variable. Due to space limitations, we only show the results in terms of TR_{10} in Table 9. The table shows that the improvement of 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, a 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.

5 DISCUSSION

Our analysis has revealed that, on average, when two methods i and j are combined using Eq.10, the generated hybrid list is often higher in quality in comparison to 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 method i.

Table 6: Comparing performance of the hybrid and individual methods in terms of MRR 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 7: 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 ↑

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 the performance. Also, if the MRR and the TR_{10} of j are higher compared to i, the combination method is more likely to have a higher MRR and a higher TR_{10} compared to 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. In fact, the combination may result in a loss of 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. 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 $\{f, g, h\}$ 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 in comparison to each other method and the number of artifacts that each IR method uniquely captured in all systems. The results are shown Table 10. In general, PMI retrieved the largest and most unique set of artifacts, therefore, was assigned a larger λ than other methods.

In summary, to answer **RQ_4**, although in most cases the performance of the hybrid methods have increased by at least a small amount in comparison to the performance of the individual methods, more performance improvement is obtained when the methods being combined returned different sets of relevant code artifacts and also achieved comparable performances individually. However, combining methods which returned a similar set of relevant artifacts did not necessarily enhance the performance. Also, combining a high performing method with a very low performing method may negatively impact the performance of the high performing method, thus, impacting the overall performance of the hybrid pair. 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 in Eq.10?** Different hybrid methods require different λ values. This value depends on the individual performance of the combined methods. Specifically, methods that are more effective individually should be assigned higher confidence levels. Our analysis also revealed

Table 8: TR_1, TR_5, and TR_{10} for Score Addition, Borda Count, and the λ-optimized approach for all combinations of IR-methods used in our experiment.

Method	System	AdditionScore			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 9: Comparing performance of the λ-optimized method to Addition Score and Borda Count in terms of TR_{10} using McNemar's Test (bold indicates significant improvement).

System	Score 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	$p < 0.05$	$p < 0.05$	$p < 0.05$	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	$p < 0.05$	$p < 0.05$	0.79	$p < 0.05$	1.00	$p < 0.05$
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	$p < 0.05$	$p < 0.05$

that an average near optimal λ can be calculated for each hybrid pair as shown in Table 4.

- **RQ_2: How effective is the hybrid approach in comparison to the individual IR methods?** Almost all methods experienced a significant improvement in performance 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.

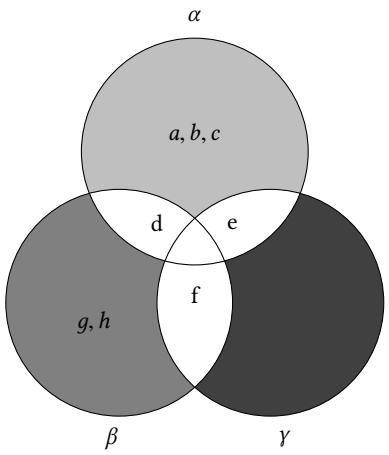

Figure 2: Example: An illustration of the calculation of the number of unique artifacts retrieved by three IR methods α, β, and γ.

Table 10: Each row shows the number of unique relevant artifacts (true positives) retrieved by each IR method in comparison to the other methods. The diagonal shows 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

- **RQ_3: How effective are the λ-optimized IR-hybrids in comparison 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 the level of 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 in more value to the hybrid pair, thus should be assigned higher confidence levels.

6 RELATED WORK AND NOVELTY

In this section, we review seminal work related to our work in this paper and we position our work within existing literature.

6.1 Related Work

The closest work to our analysis in this paper can be found in Thomas et al. [45]. The authors investigated the impact of a large space of IR configurations (e.g., code pre-processing, similarity metrics, and term weights) on the accuracy of IR-based bug localization methods. The authors reported that different parameter settings had considerable impacts on the accuracy of IR methods. The results also suggested that combining multiple IR methods, using Borda Count and Score Addition, improved the performance in all cases.

Binkley et al. [4] investigated the impact of using Learning to Rank (LtR) in IR-based software maintenance tasks. LtR is machine learning technique developed to learn how to better rank the documents retrieved using IR methods. Such technique can learn to optimize the weights of different IR features for a 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 [35]). Evaluating LtR over two common software maintenance tasks (feature location and traceability) showed that that LtR enabled statistically significant improvements in multiple performance indicators over several baseline methods.

Wang and Lo [49] 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 using thousands of bug reports from *AspectJ*, *Eclipse*, and *SWT*. The results showed that the proposed approach improved the hit rate at 5 (TR_5) and MRR over multiple systems.

Gethers et al. [16] used the λ-optimized approach to combine orthogonal IR techniques, which have been statistically shown to produce dissimilar results, to enhance the performance in IR-based requirements traceability tasks. The authors experimented with combining VSM, JMS, and Relational Topic Modeling (RTM) [6]. An empirical evaluation conducted on six software systems showed that the integrated method outperformed stand-alone IR methods as well as any other combination of non-orthogonal methods with statistically significant margins.

6.2 Novelty and Impact

Our review of seminal related work shows that existing work is mainly focused on investigating the performance of various IR methods in matching bug reports with their related buggy code artifacts (e.g. [20, 24, 38]). Other lines of research include the design, development, and evaluation of IR-based bug localization tools (e.g. [30, 40, 54]), enhancing the performance of existing IR-based bug localization methods (e.g. [4, 7, 16, 29, 45, 49, 52]), or proposing more valid measures to assess the effectiveness of these methods (e.g. [2, 22]). Our work in this paper builds upon existing work to propose a more effective solution for the problem. In particular, our work advances the state-of-the-art as follows:

- We provide further evidence that combining multiple IR methods can enhance the performance of IR-based bug localization tools. However, our analysis shows that relying

on unoptimized techniques can limit the performance of hybrid methods. Furthermore, we advance previous work [16] by empirically determining near-optimal configurations for specific combinations of methods. Such configurations can be used as reference points for developers attempting to get the best out of their IR-based retrieval engines. More accurate retrieval can help to reduce the cognitive effort required to localize bugs in large and complex software systems. This can be particularly important in Open Source environments, such as GitHub or SourceForge. In such environments, projects are maintained by distributed teams of contributors who often work on multiple active releases of the project. This search space of code, change reports, and commits, has to be initially reduced in order to minimize the information overload and help developers, especially newcomers, to find tasks that match their skill set. In fact, the ability to find bugs that are of interest has been identified as a main problem facing OSS developers [44].

- Our approach minimizes the number of configurations (a single λ knob) that developers need to work with in order to reach acceptable performance levels. Using other retrieval configurations (e.g., text pre-processing) can lead to overly complicated models (e.g., Thomas et al. identified around 3,172 configurations [45]). Furthermore, our proposed approach is unsupervised. Unlike the Machine Learning (ML) techniques proposed in [4, 49], no training of any sort is required to achieve a significant enhancement in performance. This gives our approach a practicality advantage over ML-based techniques especially that such techniques are frequently associated with hidden technical debt [42].
- Our findings highlight the value of the semantic characteristics of words (e.g., meaningful domain specific words) over their syntactic characteristics. Semantically fit words give the code its linguistic identity. IR methods such as PMI exploit this identity to establish connections between bug reports and fragments of source code. However, recent evidence revealed that most software systems exhibit a significant amount of linguistic change during their lifetime [47]. Therefore, maintaining the accuracy of IR methods require preserving the linguistic fitness of the system by constantly refactoring linguistic anomalies during code reviews.

7 THREATS TO VALIDITY

The analysis presented in this paper has several limitations that might affect the validity of the results [51]. 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 IR methods. However, since we are dealing with Object Oriented systems, each class supposedly encapsulates one functionality, thus can be treated as a separate artifact. An internal validity argument could be made about using a brute-force strategy to calibrate methods such as LSI and determine λ in Eq. 3. Other calibration strategies (e.g., Genetic algorithms) might provide a more efficient solution to the problem [23]. Such strategies become especially useful when dealing with a large number of features that make a brute-force solution infeasible.

Threats to external validity are conditions that limit the ability to generalize the results of the experiment [51]. In particular, the results of our experiment might not generalize beyond the specific experimental settings used in this paper. A potential threat to our external validity stems from the subject systems used in our analysis. Specifically, our calibration was carried out over only five systems. Therefore, further experimentation over more, and perhaps larger, datasets is necessary to ensure the generalizability of our optimized λ values.

Construct validity is the degree to which the various performance measures accurately capture the concepts they purport to measure [51]. In our experiment, there were minimal threats to construct validity as standard performance measures (MRR and TR_n), which are commonly used in bug localization research, were used to assess the performance of our investigated methods. We believe that these measures sufficiently captured and quantified the different aspects of performance we were interested in.

8 CONCLUSIONS AND FUTURE WORK

In this paper, we investigated the impact of combining different IR methods on the accuracy of static bug localization techniques. 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 almost always resulted in improvement over all performance indicators. However, the amount of improvement was highly dependent on the performance of individual IR methods. The results also showed that near optimal confidence levels can be determined for each hybrid pair of IR methods. Specifically, methods that retrieve more unique relevant artifacts individually should be assigned more confidence when combined with their less effective counterparts, otherwise, treating the combined IR methods equally can limit any potential improvement in performance.

In our future work, we will build automated solution to help calibrate IR methods in the context of large-scale distributed OSS environments. Our goal is to be able to automatically determine which IR methods, or combinations of methods, work best for different types of software systems or code retrieval tasks, and how much confidence should be assigned to each method. Furthermore, a working tool that implements our main findings in this paper will be developed. Such tool will enable us to conduct long term usability studies to gain a better understanding of our methods' scalability, usability, and scope of applicability.

ACKNOWLEDGMENT

This research is supported by the U.S. National Science Foundation (Award CCF 1821525).

REFERENCES

[1] Aharon Abadi, Mordechai Nisenson, and Yahalomit Simionovici. 2008. A Traceability Technique for Specifications. In *International Conference on Program Comprehension*. 103–112.

[2] Matthew Beard, Nicholas Kraft, Letha Etzkorn, and Stacy Lukins. 2011. Measuring the Accuracy of Information Retrieval Based Bug Localization Techniques. In *Working Conference on Reverse Engineering*. 124–128.

[3] David Binkley and Dawn Lawrie. 2010. Information retrieval applications in software maintenance and evolution. *Encyclopedia of Software Engineering* (2010), 454–463.

[4] David Binkley and Dawn Lawrie. 2014. Learning to rank improves IR in SE. In *International Conference on Software Maintenance and Evolution*. 441–445.

[5] Gerlof Bouma. 2009. Normalized (pointwise) mutual information in collocation extraction. In *Proceedings of GSCL*. 31–40.

[6] Jonathan Chang and David Blei. 2010. Hierarchical relational models for document networks. *The Annals of Applied Statistics* 4, 1 (2010), 124–150.

[7] Oscar Chaparro, Jing Lu, Fiorella Zampetti, Laura Moreno, Massimiliano Di Penta, Andrian Marcus, Gabriele Bavota, and Vincent Ng. 2017. Detecting Missing Information in Bug Descriptions. In *Joint Meeting on Foundations of Software Engineering*. 96–407.

[8] Kenneth Church and Patrick Hanks. 1990. Word Association Norms, Mutual Information, and Lexicography. *Computer Linguistics* 16, 1 (1990), 22–29.

[9] Thomas Cover and Joy Thomas. 1991. *Elements of Information Theory*. Wiley-Interscience.

[10] Nick Craswell. 2009. Mean reciprocal rank. In *Encyclopedia of Database Systems*. Springer, 1703–1703.

[11] Scott Deerwester, Susan Dumais, George Furnas, Thomas Landauer, and Richard Harshman. 1990. Indexing by latent semantic analysis. *Journal of the American Society for Information Science* 41, 6 (1990), 391–407.

[12] S. Dumais, G. Furnas, T. Landauer, S. Deerwester, and R. Harshman. 1988. Using Latent Semantic Analysis to Improve Access to Textual Information. In *SIGCHI Conference on Human Factors in Computing Systems*. 281–285.

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

[14] Merijn Van Erp and Lambert Schomaker. 2000. Variants of the Borda count method for combining ranked classifier hypotheses. In *International Workshop On Frontiers In Handwriting Recognition*.

[15] Atsushi Fujii. 2007. Enhancing Patent Retrieval by Citation Analysis. In *International ACM SIGIR Conference on Research and Development in Information Retrieval*. 793–794.

[16] Malcom Gethers, Rocco Oliveto, Denys Poshyvanyk, and Andrea De Lucia. 2011. On integrating orthogonal information retrieval methods to improve traceability recovery. In *International Conference on Software Maintenance*. 133–142.

[17] J. Huffman-Hayes, A. Dekhtyar, and S. Sundaram. 2006. Advancing candidate link generation for requirements tracing: the study of methods. *IEEE Transactions on Software Engineering* 32, 1 (2006), 4–19.

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

[19] Saket Khatiwada, Michael Kelly, and Anas Mahmoud. 2016. STAC: A tool for Static Textual Analysis of Code. In *International Conference on Program Comprehension*. 1–3.

[20] Saket Khatiwada, Miroslav Tushev, and Anas Mahmoud. 2017. Just enough semantics: An information theoretic approach for IR-based software bug localization. *Information and Software Technology* 93 (2017), 45–57.

[21] Tien Le, Richard Oentaryo, and David Lo. 2015. Information Retrieval and Spectrum Based Bug Localization: Better Together. In *Joint Meeting on Foundations of Software Engineering*. 579–590.

[22] Tien-Duy Le, Ferdian Thung, and David Lo. 2014. Predicting Effectiveness of IR-based Bug Localization Techniques. In *International Symposium on Software Reliability Engineering*. 335–345.

[23] Sugandha Lohar, Sorawit Amornborvornwong, Andrea Zisman, and Jane Cleland-Huang. 2013. Improving trace accuracy through data-driven configuration and composition of tracing features. In *Joint Meeting on Foundations of Software Engineering*. 378–388.

[24] S. Lukins, N. Kraft, and L. Etzkorn. 2008. Source Code Retrieval for Bug Localization Using Latent Dirichlet Allocation. In *Reverse Engineering*. 155–164.

[25] Anas Mahmoud and Nan Niu. 2015. On the Role of Semantics in Automated Requirements Tracing. *Requirements Engineering* 20, 3 (2015), 281–300.

[26] Christopher Manning, Prabhakar Raghavan, Hinrich Schütze, et al. 2008. *Introduction to information retrieval*. Cambridge University Press Cambridge.

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

[28] Rada Mihalcea, Courtney Corley, and Carlo Strapparava. 2006. Corpus-based and Knowledge-based Measures of Text Semantic Similarity. In *National Conference on Artificial Intelligence*. 775–780.

[29] Laura Moreno, John Treadway, Andrian Marcus, and Wuwei Shen. 2014. On the Use of Stack Traces to Improve Text Retrieval-Based Bug Localization. In *International Conference on Software Maintenance and Evolution*. 151–160.

[30] Anh Nguyen, Tung Nguyen, Jafar Al-Kofahi, Hung Nguyen, and Tien Nguyen. 2011. A topic-based approach for narrowing the search space of buggy files from a bug report. In *Automated Software Engineering*. 263–272.

[31] Rocco Oliveto, Malcom Gethers, Denys Poshyvanyk, and Andrea De Lucia. 2010. On the equivalence of information retrieval methods for automated traceability link recovery. In *International Conference on Program Comprehension*. 68–71.

[32] Thomas Ostrand, Elaine Weyuker, and Robert Bell. 2005. Predicting the location and number of faults in large software systems. *IEEE Transactions on Software Engineering* 31, 4 (2005), 340–355.

[33] Chris Parnin and Alessandro Orso. 2011. Are Automated Debugging Techniques Actually Helping Programmers?. In *International Symposium on Software Testing and Analysis*. 199–209.

[34] Nancy Pennington. 1987. Stimulus structures and mental representations in expert comprehension of computer programs. *Cognitive Psychology* 19, 3 (1987), 295–341.

[35] Jay Ponte and W Bruce Croft. 1998. A Language Modeling Approach to Information Retrieval. In *Annual International ACM SIGIR Conference on Research and Development in Information Retrieval*. 275–281.

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

[37] Denys Poshyvanyk, Yann-Gaël Guèhèneuc, Andrian Marcus, and Giuliano Antoniol. 2007. Feature Location Using Probabilistic Ranking of Methods Based on Execution Scenarios and Information Retrieval. *Software Engineering, IEEE Transactions* 33, 6 (2007), 420–432.

[38] Shivani Rao and Avinash Kak. 2011. Retrieval from Software Libraries for Bug Localization: A Comparative Study of Generic and Composite Text Models. In *Working Conference on Mining Software Repositories*. 43–52.

[39] Shivani Rao and Avinash Kak. 2013. *moreBugs: A New Dataset for Benchmarking Algorithms for Information Retrieval from Software Repositories*. Technical Report. Purdue University, School of Electrical and Computer Engineering.

[40] Ripon Saha, Matthew Lease, Sarfraz Khurshid, and Dewayne Perry. 2013. Improving bug localization using structured information retrieval. In *Automated Software Engineering*. 345–355.

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

[42] D. Sculley, Gary Holt, Daniel Golovin, Eugene Davydov, Todd Phillips, Dietmar Ebner, Vinay Chaudhary, Michael Young, Jean-Francois Crespo, and Dan Dennison. 2015. Hidden Technical Debt in Machine Learning Systems. In *International Conference on Neural Information Processing Systems*. 2503–2511.

[43] Elliot Soloway and Kate Ehrlich. 1984. Empirical Studies of Programming Knowledge. *IEEE Transactions on Software Engineering* 10, 5 (1984), 595–609.

[44] Igor Steinmacher, Marco Aurelio, Graciotto Silva, Marco Aurelio Gerosa, and David F.Redmiles. 2015. A systematic literature review on the barriers faced by newcomers to open source software projects. *Information and Software Technology* 59 (2015), 67–85.

[45] Stephen Thomas, Meiyappan Nagappan, Dorothea Blostein, and Ahmed Hassan. 2013. The impact of classifier configuration and classifier combination on bug localization. *IEEE Transactions on Software Engineering* 39, 10 (2013), 1427–1443.

[46] Peter Turney. 2001. Mining the Web for Synonyms: PMI-IR Versus LSA on TOEFL. In *European Conference on Machine Learning*. 491–502.

[47] Miroslav Tushev, Saket Khatiwada, and Anas Mahmoud. 2019. Linguistic Change in Open Source Software. In *International Conference on Software Maintenance and Evolution*. 296–300.

[48] Haoren Wang and Huzefa Kagdi. 2018. A Conceptual Replication Study on Bugs that Get Fixed in Open Source Software. In *International Conference on Software Maintenance and Evolution*. 299–310.

[49] Shaowei Wang and David Lo. 2014. Compositional Vector Space Models for Improved Bug Localization. In *International Conference on Software Maintenance and Evolution*. 171–180.

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

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

[52] Xin Ye, Razvan Bunescu, and Chang Liu. 2014. Learning to Rank Relevant Files for Bug Reports Using Domain Knowledge. In *ACM SIGSOFT International Symposium on Foundations of Software Engineering*. 689–699.

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

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