Improving Bug Localization using Structured Information Retrieval

Ripon K. Saha* Matthew Lease† Sarfraz Khurshid* Dewayne E. Perry*
*Department of Electrical and Computer Engineering
†School of Information
The University of Texas at Austin, USA
ripon@utexas.edu, ml@ischool.utexas.edu, khurshid@ece.utexas.edu, perry@mail.utexas.edu

***Abstract*—Locating bugs is important, difficult, and expensive, particularly for large-scale systems. To address this, natural language information retrieval techniques are increasingly being used to suggest potential faulty source files given bug reports. While these techniques are very scalable, in practice their effectiveness remains low in accurately localizing bugs to a small number of files. Our key insight is that structured information retrieval based on code constructs, such as class and method names, enables more accurate bug localization. We present BLUiR, which embodies this insight, requires only the source code and bug reports, and takes advantage of bug similarity data if available. We build BLUiR on a proven, open source IR toolkit that anyone can use. Our work provides a thorough grounding of IR-based bug localization research in fundamental IR theoretical and empirical knowledge and practice. We evaluate BLUiR on four open source projects with approximately 3,400 bugs. Results show that BLUiR matches or outperforms a current state-of-the-art tool across applications considered, even when BLUiR does not use bug similarity data used by the other tool.**

***Index Terms*—Bug localization, information retrieval, search**

I. Introduction

Frederick Brooks wrote that "Software entities are more complex for their size than perhaps any other human construct because no two parts are alike (at least above the statement level)" [5]. Due to this inherent complexity of software construction, software bugs remain frequent. For a large software system, the number of bugs may range from hundreds to thousands. Generally, bug fixing starts with finding relevant buggy source code. i.e., *bug localization*. However, performing this process manually for many bugs is time consuming and expensive. Therefore, effective methods for locating bugs automatically from bug reports are highly desirable.

There are two general approaches for bug localization: i) dynamically locating the bug via program execution together with such technologies as execution and data monitoring, breakpoints etc. [1]; and ii) statically locating bugs via various forms of analyses using the bug reports together with the code [15]. The dynamic approach is often time consuming and expensive. The ease of the static approach, together with its immediate recommendation, make it appealing.

In recent years, information retrieval (IR) based bug localization techniques have gained significant attention due to their relatively low computational cost and minimal external dependencies (e.g., requiring only source code and bug report in order to operate) [2]. In these IR approaches, each bug report is treated as a *query*, and the source files to be searched comprise the *document collection*. IR techniques then rank the documents by predicted relevance, returning a ranked list of candidate source files which may contain the bug. Lukins et al. [21] proposed a Latent Dirichlet Allocation (LDA) approach, while Rao et al. [29] compared a range of IR techniques: Unigram, Vector Space, Latent Semantic Analysis (LSA), LDA, Cluster Based, and various combinations. Both used a relatively small number of bugs in evaluation. Ngyuen et al. proposed *BugScout* [25], which customized LDA for bug localization. Results on several large-scale datasets showed good performance. Recently, Zhou et al. [46] proposed *BugLocator*, which combined a sophisticated TF.IDF formulation, a modeling heuristic for file length, and knowledge of previously fixed similar bugs. In a large scale evaluation of approximately 3,400 bugs over four open source projects, BugLocator showed even stronger performance than BugScout. Moreover, datasets and BugLocator's executable were made available, providing an invaluable benchmark for testing and comparing alternative IR approaches to bug localization.

Despite the empirical success of prior work, we perceive a gap today between IR community practices and techniques being applied to bug localization. For example, existing IR-based bug localization treats source code as flat text lacking structure. In fact, source code's rich structure distinguishes code constructs such as comments, names of classes, methods, and variables, etc. While ignoring such code structure simplifies the system, it also sacrifices an opportunity to exploit this structural information to improve localization accuracy. While we believe modeling source code structure is novel for bug localization, we also note that the concept of modeling document structure in IR is quite old (e.g., Google in 1998 [4] and more recent BM25F [31]).

Whereas recent prior work devised a heuristic to model program length, we discuss how the importance of length normalization was actually recognized in IR two decades ago [38] and is built-into today's baseline IR models. In the same vein, we discuss how the use of bug similarity data to improve localization is closely related to the established IR use of relevance feedback data [33]. Generalizing from this, we suspect our idea for modeling code structure is only one of the many ways in which IR-based bug location could benefit from greater interaction with the IR community. Beyond our

978-1-4799-0215-6/13 © 2013 IEEE

ASE 2013, Palo Alto, USA

technical contributions, our approach strives to forge stronger conceptual ties between ongoing work in bug localization and proven practices from the IR community.

We introduce BLUiR (Bug Localization Using information Retrieval), an automatic bug localization tool based on the concept of structured information retrieval. Rather than build BLUiR's IR indexing and retrieval system from scratch, we instead build upon an existing, highly-tuned, open source IR toolkit, Indri [40]. While we use an off-the-shelf IR tool, we simultaneously stress the importance of using it effectively, i.e., recognizing and addressing domain-specific particulars of bug localization. In our work, we extract and model code constructs like structured documents, and we show how a seemingly trivial change to how camel case identifiers are indexed yields significantly improved localization accuracy.

We evaluate BLUiR using the same large-scale benchmark on which BugLocator was evaluated. When bug similarity data is not used, the off-the-shelf IR tookit (unmodified) already exceeds BugLocator's accuracy. With our enhancements (e.g., structural modeling and indexing camel case identifiers as-is), accuracy is significantly improved further. Modeling additional bug similarity data provides yet a further gain. Finally, even if BugLocator is given bug similarity data and BLUiR is not, BLUiR still outperforms BugLocator on three of the four code repositories and matches its accuracy on the fourth. For reproducibility, data from our experiments is available online.[1]

Contributions. We present: 1) new techniques for increasing localization accuracy, particularly modeling of source code structure; 2) new state-of-the-art accuracy for bug localization on a public community benchmark, built on a proven, open source IR toolkit anyone can use; and 3) thorough grounding of IR-based bug localization research in fundamental IR theoretical and empirical knowledge and practice.

II. Background

We begin this section with a demonstrative example of IR-based bug localization. The fundamental assumption underlying these techniques is that some terms in a given bug report will be found in source files needing to be fixed for that bug. Figure 1 presents a real world bug report from Eclipse 3.1 and corresponding source code fix, taken from [46]. The Figure shows matching words (in bold font) found in both the bug report and one of the corresponding source code files that was ultimately fixed for that bug.

In IR-based bug localization, a software system's source code files represent the *document collection* to search, with each bug report being a search *query*. Finding candidate files to fix is then reduced to standard IR ranking of documents (source files) based on estimated relevance to each query (bug report). The better an IR system can interpret the bug report and source files, the more accurately it is expected to highly rank the source files needing to be fixed. While deep semantics remain elusive, shallow matching often works quite well.

[1]http://users.ece.utexas.edu/~perry/work/esel/bl/

Bug ID: 80720
Summary: **Pinned console** does not remain on top
Description:
*Open two **console views**, ... **Pin** one **console.** Launch another program that produces output. Both **consoles display** the last launch. The **pinned console** should remain **pinned.***

Source code file: **ConsoleView.java**

```
public class ConsoleView extends PageBookView
    implements IConsoleView, IConsoleListener {...
      public void display(IConsole console) {
          if (fPinned && fActiveConsole != null) { return;}
      } …
      public void pin(IConsole console) {
            if (console == null) {  setPinned(false);
            } else {
               if (isPinned()) { setPinned(false); }
               display(console);
               setPinned(true); }}}
```

Fig. 1. An example of a bug localization [46]

A. Information Retrieval (IR)

For a broad overview of IR, see [23] online. An IR system typically begins with three-step preprocessing: *text normalization*, *stopword removal*, and *stemming*. Normalization involves removing punctuation, performing case-folding, tokenizing terms, etc, ultimately defining the initial vocabulary in which queries and documents will be represented. Next, a set of extraneous terms identified in a stopword list (e.g., "to", "the", "be", etc.) are filtered out in order to improve efficiency and reduce spurious matches. Finally, stemming conflates variants of the same underlying term (e.g., "ran", "running", "run") to improve term matching between query and document.

While these three preprocessing steps are often given short shrift in describing IR approaches, they embody important tradeoffs that can significantly influence the ultimate success or failure of the retrieval model. For example, normalization can increase matches between query and document by case-folding (improving recall), but this can also introduce spurious matches as well (hurting precision). Similarly, while stopword removal can reduce unhelpful term matching (e.g., "to"), any stopword removed is almost certain to hurt matching for some particular query (e.g., "to be or not to be"). Finally, stemming will similarly increase recall by conflating variants of the same underlying term, but this may also introduce false matches. For reproducible experimentation, preprocessing methods should be fully described along with other details of IR model.

Once queries and documents have been pre-processed, documents are *indexed* by collecting and storing various statistics, such as *term frequency* (TF, the number of times a term occurs in a given document), and *document frequency* (DF, the number of documents in which the term appears). IDF refers to inverse (dampened) DF, most simply formulated as $log(\frac{N}{DF})$, where N is the number of documents in the colleciton.

A widespread misconception about TF.IDF merits particular attention. Specifically, "The TF.IDF model is often used as a baseline model for comparison with new retrieval models. However, it is not actually a well-defined model, in the sense that there are several heuristic components in the model that can affect performance significantly." [44]. For greater detail,

see [37], [32], [34]. Many studies have claimed improvement over TF.IDF using only the most naive version of the model, or simply report a TF.IDF baseline without fully specifying what TF.IDF model was actually used. In contrast, BugLocator fully-specified the TF.IDF formulation they defined [46]. However, this raises a related issue: it is not clear how their formulation differed, conceptually or empirically, to other existing, state-of-the-art TF.IDF variants. We advocate always trying simple, well-tuned, existing models first, then fully describing and motivating whatever TF.IDF variant is used.

III. Presence of Source Code Terms in Bug Reports: An Empirical Study

The success of IR-based bug localization is dependent on effectively matching the bug report to source files needing to be fixed. As discussed in Section II, even preprocessing issues can significantly impact IR accuracy. The classic IR challenge lies in effectively recognizing important terms in the query and document, and assigning each greater weight for matching. With regard to text length, long queries (e.g., when using the bug report's `description` field) can obscure key search terms [18]. Document length also merits special attention [38]. Both topics are further discussed in Section IX-C.

Another classic IR approach distinguishes and separately models different fields when text is structured [4], [31]. For example, while searching documents, Google considers page title, different anchor texts, and body separately [4]. We investigate this structured approach to IR-based bug localization. With queries, a bug report contains separate `summary` and `description` fields; whereas the summary provides essential keywords, the description is more verbose with additional terms. As discussed in the next section, source code files are even more structured. We perform preliminary analysis here to assess the degree to which source code terms appear in bug reports, potentially providing an opportunity for better IR.

We distinguish six types of terms. Query terms come from different bug report fields: the concise `summary` and verbose `description`). Parsing source code structure also lets us distinguish four different document fields: `class`, `method`, `variable`, and `comments`. These fields are extracted by constructing and traversing the abstract syntax tree (AST) of the subject program (Section IV-A). For each bug report, we separately search for terms from each document field in source files that were fixed for the corresponding bug. We collect two separate sets of statistics: matching terms "as is" in their original form vs. splitting identifier names based on the camel case heuristics and searching for each token.

To illustrate, for the given example in Figure 1, first we search `ConsoleView`, and then the separate terms `console` and `view`, in both the bug summary and bug description. For each search, we exclude those tokens that either are stop words or have less than three characters. For example, if a variable name is `isBalancedTree`, we do not search for "is".

Table I provides empirical evidence that terms from source files to fix in AspectJ are present in the corresponding bug reports. Each entry represents the number of bug reports in which different term types (class, method, variable, or comment) were found. For each bug report section (`summary` vs. `description`), we count the number of bug reports containing an exact match or token match for at least one of the files to be fixed. For example, the first two numbers in the "class" row of Table I represent that in 27 bug reports in AspectJ, at least one of the class names of the fixed files was present as-is in the bug summary, whereas in 101 bug reports at least one of the class name's split tokens was present. From the Table, we see that although summary contained only the 3% of total terms in the bug report, at least one of the (class, method, variable, comment) terms was found in (35%, 72%, 43%, 82%) of the bug summaries, respectively. Similarly, although class name is typically a combination of 2-4 terms per source code file, they are present in more than 35% of the bug summaries and 85% of the bug descriptions. Furthermore, the exact class name is present in more than 50% of the bug descriptions. We can observe a similar phenomena for method names as well.

TABLE I
Presence of Different Term Types in Bug Reports for AspectJ

Term Type	Summary		Description	
	Exact match	Token match	Exact mach	Token match
Class	27 (9.44%)	101 (35.31%)	148 (51.74%)	244 (85.31%)
Method	43 (15.03%)	205 (71.67%)	187 (65.38%)	277 (96.85%)
Variable	107 (37.41%)	125 (43.70%)	230 (80.42%)	252 (88.11%)
Comments	N/A	235 (82.16%)	N/A	278 (97.20%)

While the bug report description vs. summary has many more matches, the more verbose `description` likely matches many irrelevant terms as well. Similarly, Table I only shows matches from the source files needing to be fixed. The bug reports also include terms matching many other source files not needing to be fixed. Consequently, this Table provides suggestive rather than conclusive evidence for our approach; evaluation later in the paper will demonstrate the empirical effectiveness of modeling this information. We intentionally restrict the analysis here to AspectJ only, reserving the other three source code repositories for later blind evaluation to maximize generality of our findings.

IV. Approach

In the previous section, we showed that important program constructs such as class names and method names are present in many bug reports and thus might be effectively used to improve bug localization. This section describes our structured IR-based approach for localizing bugs.

A. BLUiR Architecture

Figure 2 shows the overall architecture of BLUiR. First BLUiR takes as input the source code files in which we would like to localize the bugs. Next, it builds the abstract syntax tree (AST) of each source code file using Eclipse Java Development Tools (JDT), and traverses the AST to extract different program constructs such as class names, method names, variable names, and comments. Then BLUiR tokenizes all the identifier names and comment words, as described in Section IV-B. This information for each source file is then stored as a structured XML document.

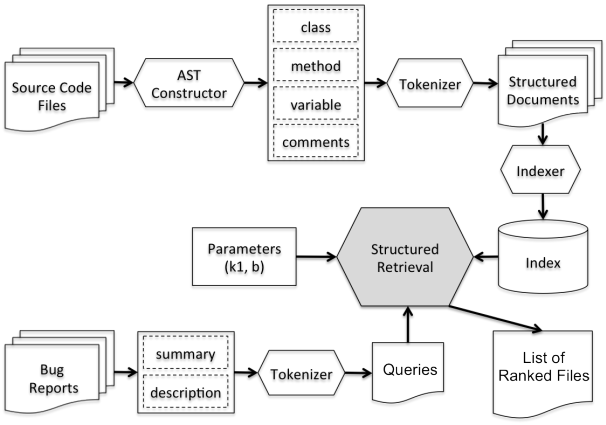

Fig. 2. BLUiR Architecture

Reducing bug localization to a standard IR task enables us to exploit a wealth of prior theoretical and empirical IR methodology, providing a robust foundation for tackling bug localization. We adopt the Indri toolkit [40] for efficient indexing and developing our retrieval model. After XML documents are created above, they are handed off to Indri for stopword removal, stemming, and indexing. Note that we used the default stopword list provided with Indri.

Each bug report is similarly tokenized, then handed off to Indri for stopping, stemming, and retrieval (Section IV-C).

B. Source Code Parsing & Term Indexing

In comparison to prior approaches, we make two improvements in our preprocessing. First, prior work has indexed all source code terms except English stopwords and programming language keywords. However, some keywords like `String` or `Class` are used in identifier names and may be found in bug reports. For example, in `AspectJ`, many identifiers use Java language keywords, e.g., `if`, `else`, etc. Therefore, instead of pruning all language keywords, we instead build the Abstract Syntax Tree (AST) of each source file and extract all identifier names (class name, method name, variable name etc.). In this way, we exclude language keywords without losing their presence in identifiers.

Secondly, identifiers are typically split into tokens for indexing to improve recall. Dit et al. [8] compared simple camel case splitting to the more sophisticated *Samurai* [10] system and found that both performed comparably in concept location. We therefore adopt camel case splitting for its simplicity. However, since our analysis in Table I reveals that full identifiers are often present in bug reports in the form of execution traces of exceptions, test cases or code snippets, we index full identifiers as well as split tokens. Although it is a very simple extension, we will see that it yields significant improvement.

C. Retrieval Model

As discussed in Section II-A, TF.IDF is not actually a well-defined model, and different TF.IDF variants can achieve vastly different empirical performance in practice. We adopt Indri's built-in TF.IDF formulation (from its parent project *Lemur*), based upon the well-established BM25 (Okapi) model [32]. This TF.IDF model has been rigorously evaluated over a decade of widespread use in IR. We elaborate below.

Assume that a document and a query are represented by a weighted term frequency vector $\vec{d}$ and $\vec{q}$ respectively of length n (the total number of terms or the size of vocabulary).

$$\vec{d} = (x_1, x_2, ..., x_n) \quad (1)$$

$$\vec{q} = (y_1, y_2, ..., y_n) \quad (2)$$

Each element of x_i of $\vec{d}$ represents the frequency (count) of term t_i in document d (similarly, y_i in query $\vec{q}$).

Generally, in a vector space model, query and document terms are weighted by a heuristic TF.IDF weighting formula instead of only their raw frequencies. Inverse document frequency (IDF) diminishes the weight of terms that occur very frequently in the document set and increases the weight of terms that occur rarely. Weighted vectors for $\vec{d}$ and $\vec{q}$ are thus:

$$\vec{d_w} = (tf_d(x_1)idf(t_1), tf_d(x_2)idf(t_2), ..., tf_d(x_n)idf(t_n)) \quad (3)$$

$$\vec{q_w} = (tf_q(y_1)idf(t_1), tf_q(y_2)idf(t_2), ..., tf_q(y_n)idf(t_n)) \quad (4)$$

Given a collection C of source files, the simplest, classic IDF formulation for term t is given by $idf(t_i) = log\frac{N}{n_t}$, where N is the total number of documents in C and n_t is the number of documents with term t. In the simplest TF-IDF model, we would simply multiply this value by the term's frequency in document d to compute the TF-IDF score for (t, d), then sum over all terms in the query to arrive at the d's TF-IDF score.

As mentioned above, however, actual TF-IDF models used in practice differ greatly from this for improved accuracy [37], [32]. We adopt Indri's TF.IDF model [44], which is summarized below.

To begin, the IDF value is smoothed as follows to avoid division by zero, which would otherwise occur whenever a particular term appears in all documents: $idf(t_i) = log\frac{N+1}{n_t+0.5}$.

The *document*'s tf function is computed by Okapi:

$$tf_d(x) = \frac{k_1 x}{x + k_1(1 - b + b\frac{l_d}{l_C})} \quad (5)$$

where k_1 is a tuning parameter (≥ 0) that calibrates document term frequency scaling. The *term frequency* value quickly saturates for a small value of k_1, whereas, a large value corresponds to using raw term frequency. b is another tuning parameter between 0 and 1, which is the document scaling factor. Recall our earlier discussion of BugLocator introducing a heuristic for modeling document length, whereas this is already built into IR models today (Section I). Here, when the value of b is 1, the term weight is fully scaled by the document length. For a zero value of b, no length normalization is applied. l_d and l_C represents the document length and average document length for the collection respectively.

The *query*'s TF function tf_q is defined similarly tf_d though $b = 0$ is fixed since the query is fixed across documents being compared, and thus normalization of query length is unnecessary:

$$tf_q(y) = \frac{k_3 y}{x + k_3} \quad (6)$$

In Equation, 6, the value of k_3 is fixed to 1000 to obtain almost the raw query term frequency because in query the

probability of having a term many times is rare. Now the similarity score of document $\vec{d}$ against query $\vec{q}$ is given by Equation 7.

$$s(\vec{d},\vec{q}) = \sum_{i=1}^{n} tf_d(x_i) tf_q(y_i) idf(t_i)^2 \quad (7)$$

D. Incorporating Structural Information

The TF.IDF model presented in Equation 7 does not consider source code structure (program construct)–i.e., each term in a source code file is considered having the same relevance with respect to the given query. Therefore, important information like class names and method names often get lost in the relatively large number of variable names and comments terms due to the term weighting function (Equation 5). Therefore, if a source code file with class name "A" also contains 10 other variable names having the term "A", then the class name "A" does not add much weight. Thus, if there is a bug report related to class "A", it will rank another file higher if the file has the term "A" more than 11 times even in the local variable names or comments. Our proposed model distinguishes different code constructs to overcome this problem.

As we described in Section III, we distinguish two alternative query representations coming from different fields of the bug report (the *summary* and the more verbose *description*). Parsing source code structure also lets us distinguish four different document fields: *class, method, variable, comments*. To exploit all of these different types of query and document representations, we perform a separate search for each of the eight (query represent, document field) combinations and then sum document scores across all eight searches.

$$s'(\vec{d},\vec{q}) = \sum_{r \in Q} \sum_{f \in D} s(d_f, q_r) \quad (8)$$

where r is a particular query representation and f is a particular document field. The benefit of this model is that terms appearing in multiple document fields are implicitly assigned greater weight, since the contribution from each term is summed over all fields in which it appears. While our method of integrating structural information is quite simple, more sophisticated methods for integrating structural information exist and could be explored in future work, e.g., doing a weighted combination rather than a simple sum, or better yet, weighting term frequencies rather than document fields to better control for term frequency saturation [31].

V. Evaluation Setup

A. Data Set

We have used the same dataset that Zhou et al. [46] used to evaluate BugLocator. This dataset contains 3,379 bug reports in total from four popular open source projects–Eclipse, AspectJ, SWT, and ZXing along with the information of fixed files for those bugs. Table II describes the dataset in more detail. Since we would like to compare BLUiR with BugLocator, the same dataset allows us to get comparable results. Among the four subject systems in the dataset, we always use AspectJ for learning purposes (to tune the parameters) so that we do not overfit our retrieval model. We chose AspectJ system as training dataset because it has 298 bugs, which is neither too large nor too small. We have also compared our results with a similar version of dataset (for AspectJ and Eclipse) that were used in evaluating BugScout (Table VII).

TABLE II
Details of Benchmark

Project	Description	Period	#Bugs	#Files
SWT 3.1	Widget toolkit for Java	10/04-04/10	98	484
Eclipse 3.1	Popular IDE for Java	10/04 03/11	3075	12863
AspectJ	Aspect-oriented extension to Java	07/02-10/06	286	6485
ZXing	Barcode image processing library for Android	03/10-09/10	20	391

B. Evaluation Metrics

Since an IR system's value is in direct proportion to how well it serves its users, the design and selection of appropriate evaluation metrics has been a topic of considerable study in IR. We should select a sufficient yet minimal set of metrics to ensure what we measure provides an appropriate and comprehensible yardstick for assessing the most pertinent aspects of performance. We err on the side of excess and comparative evaluation, including all five metrics as Zhou et al. [46]; other systems we compare to use a subset of these metrics. All metrics are based on gain rather than loss (larger values indicate better performance).

Recall at Top N: This metric reports the number of bugs with at least one buggy source file found in the top N (= 1, 5, 10) ranked results (once the first buggy file is located, it may become easier for the developer to find the rest). Since we are only considering the top few ranks, and only requires finding one of the buggy files per bug, this metric emphasizes early precision over total recall.

Mean Reciprocal Rank (MRR): Like "Recall at Top N", MRR emphasizes early precision over recall. The reciprocal rank for a query is the inverse rank of the first relevant document found. MRR is the reciprocal rank averaged over all queries:

$$MRR = \frac{1}{|Q|} \sum_{i=1}^{|Q|} \frac{1}{rank_i} \quad (9)$$

Mean Average Precision (MAP): MAP is by far the most commonly used, traditional IR metric. It takes all the faulty files into account with their ranks. Therefore, MAP emphasizes recall over precision, and is favored in scenarios in which users will go deep in a ranked list to find many relevant results. The Average Precision of a single query is computed as:

$$AP = \sum_{k=1}^{M} \frac{P(k) \times pos(k)}{number\ of\ positive\ instances} \quad (10)$$

where k is the rank, M is the number of retrieved source files, and $pos(k)$ is a binary indicator of whether or not the item at rank is a buggy file. $P(k)$ is the precision at the given cut-off rank k. The MAP for a set of queries is simply the mean of the average precision values for all queries.

Note that all metrics above compute an arithmetic mean over the query set to measure average performance. This may not

TABLE III
EFFECT OF DIFFERENT STEMMERS AND PARAMETERS ON ASPECTJ

Term Weighting	Stemmer	Top 1	Top 5	Top 10	MAP	MRR
	none	29	93	134	0.12	0.22
$k_1 = 1000$, $b = 0$	Krovetz	29	97	134	0.12	0.22
	Porter	27	99	135	0.12	0.21
$k_1 = 1.2$, $b = 0.75$	Krovetz	77	130	162	0.20	0.37
$k_1 = 1.0$, $b = 0.3$	Krovetz	79	131	168	0.20	0.37

be appropriate if developer satisfaction is driven by worst-case performance rather than average performance, in which case a geometric mean may be more appropriate. Figure 3 presents a per-query analysis of results, inspecting the performance of each query instead of only the average.

C. System Tuning

As we described in Section V-A, we use AspectJ as training dataset to choose stemmer and tune two parameters of our model: the term weight scaling parameter k_1 and the document normalization parameter b. Table III compares system performance with no stemmer vs. using two popular stemmers: Krovetz and Porter. For this experiment (only), we use approximately raw TF.IDF with $k_1 = 1000$, and $b = 0$. We observe no significant difference among the three methods. In prior work, Hill et al. [13] also observed that no single stemmer is better for all kinds of queries. While we choose Krovetz somewhat arbitrarily, as the more conservative of the two stemming algorithms, closer analysis here appears to be warranted to provide a fuller explanation.

Table III shows results of tuning k1 and b. These experiments exclude modeling of source code structure. Traditional wisdom is to set $k_1 = 1.2$ and $b = 0.75$. However, since bug localization is different from traditional text retrieval, we did a linear sweep of all values between 0:2 for k_1 and 0:1 for b (with step-size 0.1), selecting $k_1 = 1.0$ and $b = 0.3$ as optimal.

VI. RESULTS

This section presents the evaluation results of BLUiR while performing bug localization on the four subject systems described in Table II. We mainly answer five research questions that show the effectiveness of different improvements that we made in developing BLUiR, and compare the results of BLUiR with other information retrieval models and tools.

RQ1: Does indexing the exact identifier names improve bug localization? In section III, we observed that in many bug reports of AspectJ, different kinds of source code entity names (e.g. class name, method name) are present exactly as-is. In this research question, we investigate the effectiveness of adding full identifier names as well as tokenized identifiers to the index. Our experiments in this section exclude source code structure. Results are reported for all four subject systems, first with only the tokenized identifier names and then with the combination of tokenized and full identifier names.

Table IV presents the result of indexing both. The evaluation results show that the addition of exact identifier names improved the accuracy for three of four subject systems. In Eclipse, using exact identifier names, BLUiR localized 217 (7.05%) more bugs in Top 1 file, whereas, the increases are

TABLE IV
EFFECT OF INDEXING FULL IDENIFIER NAMES

System	Indexed	Top 1	Top 5	Top 10	MAP	MRR
SWT	Tokens	29	72	82	0.41	0.48
	Both	37	71	84	0.47	0.54
Eclipse	Tokens	529	1121	1415	0.20	0.27
	Both	746	1378	1647	0.26	0.34
AspectJ	Tokens	79	131	168	0.20	0.37
	Both	87	147	175	0.22	0.41
ZXing	Tokens	8	11	12	0.35	0.48
	Both	7	11	12	0.35	0.45

TABLE V
EFFECT OF MODELING SOURCE CODE STRUCTURE

System	Structure	Top 1	Top 5	Top 10	MAP	MRR	ET/$Q_i(s)$
SWT	N	37	71	84	0.47	0.54	0.05
	Y	54	75	85	0.56	0.65	0.21
Eclipse	N	746	1378	1647	0.26	0.34	0.44
	Y	952	1636	1933	0.32	0.42	5.45
AspectJ	N	87	147	175	0.20	0.37	0.57
	Y	92	146	173	0.24	0.41	4.22
ZXing	N	7	11	12	0.35	0.45	0.08
	Y	8	13	14	0.38	0.49	0.25

8.16% and 2.79% for SWT and AspectJ respectively. The consistently higher MAP and MRR for the first three subject systems show the overall improvements of the ranking due to adding exact identifiers. In ZXing, the Top 1 and MRR metrics were a little bit lower than the traditional one, while other metrics were exactly the same. However, it is very difficult to derive any useful conclusions from ZXing because the bug dataset has only 20 bugs for this subject system.

RQ2: Does modeling source code structure help improve accuracy? In section III, we argued that source code structure, i.e., distinguishing different code constructs could be effectively used to find more important terms in both source code and bug report and thus improve the overall bug localization accuracy. In this research question, we investigate whether this improves the accuracy of bug localization and, if so, how much. To this end, we ran BLUiR on all the subject systems to localize bugs with and without modeling source code structure.

Table V results show that in most cases, BLUiR performed better in terms of all the metrics when it considered different program constructs. More specifically, structured retrieval is more effective for Top 1. Using structured retrieval, BLUiR localized 17 (17.35%), 206 (6.70%), 5 (1.74%), and 1 (5%) more bugs in SWT, Eclipse, AspectJ, and ZXing respectively within top 1 file. In AspectJ, for Top 5 and Top 10, BLUiR localized a few less bugs using structured retrieval. However, the high MAP and MRR shows that the overall ranking is much better when BLUiR used structured retrieval. Further qualitative analysis is presented in Section VII.

Runtime Overhead: Since structured information retrieval involves more computation than the normal text retrieval, the runtime overhead of structured information retrieval is expected to be higher. To investigate this overhead, we computed the average execution time per query (ET/Qi) of BLUiR both for traditional retrieval and structured retrieval (Table V). The results show that structured retrieval is more costly and the overhead depends on the size of *document collection*. The

specific cost varies from about 3x to 12x. However, since all the execution times are less than 6 seconds, the added cost from the developers perspective is negligible. Moreover, structured information retrieval due to its higher accuracy may save quite a bit of the developers time overall.

RQ3: Does BLUiR outperform other bug localization tools and models? While evaluating BugLocator, Zhou et al. [46] compared their model with other prior work, and showed that BugLocator consistently performed best. We therefore compare BLUiR to with BugLocator, which is, to the best of our knowledge, the most accurate tool presently.

Table VI shows results of BLUiR and BugLocator for the given dataset, using and without using similar bug report data. BugLocator results are copied verbatim from [46]. It should be noted that we use the same datasets used to evaluate BugLocator. In this section, we restrict our discussion to results without using bug similarity data.

Comparing each metric of each system, we can see that, for ZXing, the results produced by both tools are almost the same. As we explained earlier, it is very difficult to derive any useful conclusions from ZXing because the bug dataset has only 20 bugs for this subject system. However, looking into the results of other systems, which have more bug reports (98 for SWT, 3075 for Eclipse, and 286 for AspectJ), we can clearly see the that BLUiR outperformed BugLocator by a great margin. BLUiR localized 23 (23.47%) more bugs in SWT, 203 (6.60%) more bugs in Eclipse, and 27 (9.44%) more bugs in AspectJ ranked within the top 1 file. The same trend is observed for other metrics as well. The consistently higher MAP and MRR for BLUiR also suggest that the overall ranking of the buggy files produced by BLUiR are better than that of BugLocator.

Now we investigate the number of queries for which BLUiR actually performed better than BugLocator because the higher number of bugs located in top 1, 5, and 10 files retrieved by BLUiR than that of BugLocator does not necessarily mean that BLUiR performed well for all queries. **Figure 3** shows per-query performance of BLUiR compared to BugLocator on SWT, where X axis represents the query number and Y axis represents the difference between the best rank of the buggy files by BLUiR and that of BugLocator. The negative value represents the query where BugLocator performs better than BLUiR. We can see that for 47 queries BLUiR performed better, for 14 queries BugLocator performed better, and for 35 queries both tools perform exactly the same. This results suggest that BLUiR performed better than BugLocator for most of the queries. Interestingly, we also observe that for 12 bug reports BLUiR improved the rank of buggy files by more than 10 positions, whereas it is only two where BugLocator improved the rank by more than 10 positions (64 and 85 positions). As a result, BLUiR places more buggy files within top 1, 5, and 10 files in the rank list than BugLocator.

This analysis required access to per-query results from BugLocator, made possible by its executable being publicly available. Unfortunately, it crashed when run on the other three collections, and we could not reach the authors for assistance in time for this submission. This analysis is therefore limited to SWT only.

TABLE VI
BLUiR VS BUGLOCATOR

System	Method	SB	Top 1	Top 5	Top 10	MAP	MRR
SWT	BugLocator	N	31	64	76	0.40	0.47
	BLUiR	N	54	75	85	0.56	0.65
	BugLocator	Y	39	66	80	0.45	0.53
	BLUiR	Y	55	75	86	0.58	0.66
Eclipse	BugLocator	N	749	1419	1719	0.26	0.35
	BLUiR	N	952	1636	1933	0.32	0.42
	BugLocator	Y	896	1653	1925	0.30	0.41
	BLUiR	Y	1013	1729	2010	0.33	0.44
AspectJ	BugLocator	N	65	117	159	0.17	0.33
	BLUiR	N	92	146	173	0.24	0.41
	BugLocator	Y	88	146	170	0.22	0.41
	BLUiR	Y	97	150	176	0.25	0.43
ZXing	BugLocator	N	8	11	14	0.41	0.48
	BLUiR	N	8	13	14	0.38	0.49
	BugLocator	Y	8	12	14	0.44	0.50
	BLUiR	Y	8	13	14	0.39	0.49

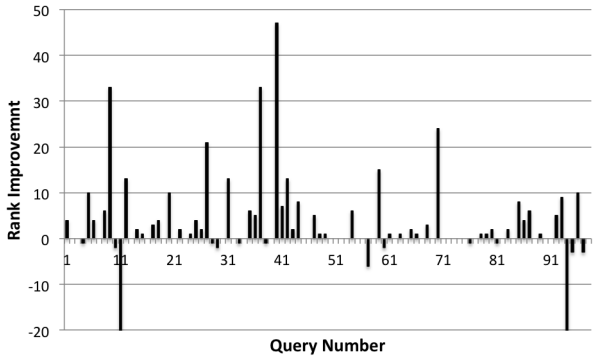

Fig. 3. Query wise comparison of BugLocator and BLUiR for SWT

We also compare our results to *BugScout* [25] and BugLocator in **Table VII**. To evaluate BugScout, Nguyen et al. used AspectJ and Eclipse as their subject systems. Since our datasets are not exactly the same as theirs, we present the differences between two datasets in terms of number of bugs in the Table. We have also interpreted the recall at Top 1, Top 5, and Top 10 of BugScout results from a Figure in their paper, which may slightly differ from their actual value. Results show that BLUiR outperformed BugScout consistently.

Recently, Sisman and Kak [39] incorporated version histories in IR-based bug localization. They proposed two models, namely the *Modification History based Prior* and the *Defect History based Prior* models, to estimate a prior probability for each file in a project having bugs. They used these priors to rank documents in addition to different models. Based on

TABLE VII
COMPARISON OF BUGSCOUT AND BUGLOCATOR WITH BLUiR

System	Description	BugScout	BugLocator	BLUiR
AspectJ	Number of Bug Reports	271	286	286
	Top 1	11%	23%	32%
	Top 5	26%	41%	51%
	Top 10	35%	56%	60%
Eclipse	Number of Bug Reports	4,136	3,075	3,075
	Top 1	14%	24%	31%
	Top 5	24%	46%	53%
	Top 10	31%	56%	63%

a case study on AspectJ, they showed that version histories improved the MAP as much as 30%. However, without considering any version history information, BLUiR performed (MAP: 0.2396) better than their best results (MAP: 0.2258).

RQ4: Does our approach compensate for the lack of similar bug information? Although the performance of BugLocator improved a lot after using similar bug fixes information, one of our main objectives is improving bug localization without using the similar bug information, since most real world projects do not explicitly have that information. In addition, reconstructing the similar bug fix information is also not a trivial task. Therefore, we investigate how BLUiR performs in locating bugs compared to BugLocator even when BugLocator uses similar bug information. The comparative results presented in Figure VI show that in most cases, the MAP and MRR of BLUiR are higher than that of BugLocator, which indicates that BLUiR overall performed better even when BugLocator considered similar bug information. For example, BLUiR localized 17 more bugs in SWT, 56 more bugs in Eclipse, and 4 more bugs in AspectJ than BugLocator within top 1 file.

We were also curious to see if we could capture those bugs that were localized by BugLocator using similar bug information. To this end, we ran BugLocator on SWT using ($\alpha = 0.2$) and not using ($\alpha = 0$) similar bug information. We found 10 such bugs in total, that have been placed within top 1, 5, or 10 files by BugLocator after using similar bug information. Interestingly, we found that BLUiR could localize all the bugs without using similarity information.

RQ5: Does similar bug fix information further improve our model? We implemented the same technique for incorporating similar bug information to BLUiR that Zhou et al. [46] did in developing BugLocator. By comparing the results of BLUiR in Table VI, we can see that the similar bug information further improved our results. For example, it helps BLUiR localize 61, 93, and 77 more bugs ranked in the top 1, 5, and 10 files respectively for Eclipse project. It also improved 1 bug localization in SWT and 5 in AspectJ within top 1 file. However, the overall improvement due to using similar bug information was not as large as that of BugLocator. Therefore, here we can conclude that BLUiR can compensate for the lack of similar bug information partially because we already localized many bugs without using similar bug fix information, which were only localized by BugLocator using similar bug information. BLUiR can also make use of the similar bug fix information to improve the model further, if it is available.

Other Results. We briefly report preliminary experiments with pseudo-relevance feedback (PRF, Section IX-B) using Indri. The primary advantage of using PRF is that we do not need to know any prior information (e.g., similar bugs) about relevant documents while running query. In PRF mode, Indri basically performs the general retrieval first, and then augments the original query by taking the m most frequent words from top r documents. There is also a tuning parameter α for weighting original query and augmented terms. Finally, the augmented query is run again to get the final rank list. We experimented with different values of m, n, and α, but did not observe improved accuracy. In future work, we would like to explore this idea further.

VII. Qualitative Analysis

The previous section presented quantitative results showing BLUiR's improvement on average over BugLocator. In this section, we dig into several queries in detail to better understand why BLUiR performs better in most cases. Consider SWT bug report #87676:

Summary: *Double-click only works on a tree's column0*

Description: *Using the log view as an example, double-clicking on column0 brings up the event dialog as it should. double-clicking on column1, column2 results in no notification to our double-click listener.*

For this bug only one file was fixed, and that is `org.eclipse.swt.widgets.Tree.java`. By reading the bug report and seeing the file name of the fixed file, one might think at a glance that this file can be identified easily. However, identifying a buggy file in a real world project is not that easy, especially where there are many other such similar files. For example, in SWT there are at least 10 other files (TreeColumn.java, TreeEvent.java, TreeListener.java, TreeAdapter.java, TreeItem.java etc.) that deal with `tree` and have this word in their file names. Thus, for a developer who did not originally implement the functionality of tree might think TreeColumn and TreeListener are more important because the bug report contains the words `column` and `listener`. Furthermore, the bug report has some other words such as `double click`, `event`, `dialog`, which are contained many times in more than 30 other files such as Text.java, Widget.java, Button.java, and so on. Therefore, finding the desired files from the IR perspective is also very challenging. Relying on only length of the files is certainly not the solution of this problem.

As a result BugLocator placed the file, Tree.java, at 50th position in the rank list. Fortunately, BLUiR first performs all the field retrievals using both the bug summary and the bug description, and then aggregates all the scores to finally rank all the source code files. This results in the summary words (e.g. `tree`) being used more advantageously. Furthermore, documents have search words (e.g. `column, double click`) spread over more fields produce better results than documents having search words found in one field. In this way, BLUiR emphasizes on more important words in the documents. As a result, BLUiR placed Tree.java at 3rd position in the rank list. In this way, BLUiR improved the rank of buggy files by more than 10 positions for 12 bug reports (e.g., bugs #78856, #79419, #83262, and so on).

VIII. Threats to Validity

This section discusses the validity and generalizability of our findings. In particular, we discuss Construct Validity, Internal Validity, and External Validity.

Construct Validity. We used two artifacts of a software repository: source code and bug reports, which are generally well understood. Our evaluation uses the same benchmark dataset of bug reports and source code shared by Zhou at

al. [46], enabling fair comparison and reproducible findings. Metrics used for evaluation match those of Zhou at al. and other prior work, are standard in IR, and are straightforward to compute. Therefore, we argue for a strong construct validity.

Internal Validity. We utilize program constructs to rank source code documents with respect to a given bug. Since we have used all the subject systems written in Java, these are mainly object-oriented (OO) constructs such as class names, method names, and so on. In this sense, our approach is language dependent. We expect our system could be easily adapted to other OO languages (future work).

Since we are matching terms between bug reports and source code, we assume meaningful identifier names and inclusion of comments, consistent with programming best practices. That said, poorly written source code would make bug localization more difficult (for both IR or non-IR approaches). Similarly, we also depend upon the quality of bug report, and poorly written reports would likely also hurt IR and non-IR methods. Our structural modeling approach matching source code terms in bug reports likely benefits significantly from the bug reports having been written by developers knowledgeable of the underlying source code. Bug reports written by end-users would likely show a far less pronounced effect.

We have used the same dataset as Zhou et al. [46]. While the possibility exists of errors in their data, this seems quite low since they have manually validated the dataset. Also, three of our four subject systems represent system-specific projects. As Hyrum et al. [41] noted, system-domain software may have its own set of development biases. Therefore, we may not capture some unique concerns, which are only present in the software development targeted toward other domains.

External Validity. We have used only four subject systems in our experiment and all of them are open source projects. Although, they are very popular projects, our findings may not be generalizable to other open source projects or industrial projects. However, to maximize generalizability of findings and minimize risk of over-fitting, we developed and tuned BLUiR on only one subject system (AspectJ), reserving the remaining three systems for final blind evaluation. This risk of insufficient generalization could be mitigated by expanding the benchmark to include more subject systems (both open source and industrial). This will be explored in our future work.

IX. Related Work

A. Automatic Bug Localization

Automatic bug localization or automatic debugging has been an active research area for over two decades [36], [35]. Existing techniques can be broadly categorized into two categories: dynamic [1] and static [15]. Generally, dynamic fault localization techniques can localize bugs very precisely (such as at statement level). However, they require a test case suite and need to execute the program for gathering passing and failing execution traces. Furthermore, the approaches are computationally expensive. Spectrum based fault localizations [1], [17], [19], dynamic slicing [45], delta debugging [43] are some of the well known techniques in this category.

Static approaches, on the other hand, do not require any program test cases or execution traces. In most cases, they need only program source code and bug reports. They are also computationally efficient. The static approaches can be also divided into two categories: i) program analysis based approaches ii) IR-based approaches. FindBug [15] is a popular bug localization tool based on static program analysis that can detect bugs by identifying buggy patterns frequently happened in practice. Therefore, FindBug does not even need a bug report. However, it cannot detect semantic bugs.

B. Information Retrieval (IR)

While historical arguments debated which of three traditionally-dominant IR paradigms was best (TF.IDF [34], the "probabilistic approach" known as BM25 (Okapi) [32], or more recent language modeling [26], all three approaches have been shown theoretically to utilize the same underlying textual features, and empirically to perform comparably when well-tuned [11]. Consequently, while one formalism or another might make it easier to integrate useful additional features, use of one formalism or another is not particularly important when it simply comes to baseline IR performance.

In contrast with shallow "bag-of-words" models, research has also explored deeper methods matching "concepts" (often poorly defined). While latent semantic indexing (LSI) induces latent concepts, it is rarely used in practice today due to errors in induced concepts introducing more harm than good. While a probabilistic variant of LSI has been devised [14], its probability model was found to be deficient. This led to now ubiquitous latent dirichlet allocation (LDA) modeling [3]. While many studies have shown LDA can usefully infer latent topics underlying a document collection, LDA is both computationally expensive and operates without reference to the input query. It has been shown that far simpler IR models based on pseudo-relevance feedback (PRF) can efficiently induce better topics on the fly for each query, tailored to the query vs. query-independent LDA topics [42]. Consequently, LDA models appear less useful for IR than simpler models until this fundamental problem can be meaningfully addressed.

One issue considered in this paper was how to best utilize multiple representations of the same bug report (i.e., its `summary` and `description`). While the summary is very succinct and likely provides the most important keywords, it may lack other terms useful for matching (suggesting high precision but possibly low recall). In contrast, the more verbose description may contain many other useful terms to match, it likely contains a variety of distracting terms as well. This is a very well-known problem in traditional IR [18]. For over two decades, data from the Text REtrieval Conference (TREC) has provided queries at three levels of verbosity, with researchers devising various methods to maximally exploit these different representations. For example, simply concatenating the two representations together provides an easy way to emphasize keywords while also including more verbose terms as well. In this paper, our method of performing separate summary and description searches and summing results is roughly equivalent to such concatenation. Future work could explore a wide

variety of more sophisticated IR methods for exploiting these alternative query representations with varying verbosity.

C. IR-based Bug Localization

The value of document length normalization was recognized in IR nearly two decades ago [38]. Empirical data compared the length of documents predicted relevant by TF.IDF vs. the length of actual relevant documents, showing that traditional IR models are actually biased against longer documents. An empirical correction for this bias was developed, it was realized that this correction was already built-into BM25, and it has been further shown that IR's language modeling paradigm performs implicit length normalization as well.

Several prior studies have investigated use of bug similarity data in order to improve localization accuracy [46], [6]. This idea can be seen as a close cousin to long-established methods for incorporating relevance feedback (RF) data in IR [33]. While RF exploits the fact that knowing one or more documents relevant to the current query makes it much easier to find other relevant documents, this knowledge is often seldom available in practice. A "trinity" of related variants has been theoretically and empirically established, showing that similar queries should retrieve similar documents (and vice-versa), and that similar documents should receive similar relevance scores for the same query (*score regularization*) [7]. In fact, the idea that the same documents should be relevant for similar queries provides the foundation for search community question and answer forums today [16]. Consequently, while use of bug similarity data for localization represents a very valuable adaptation of RF methods from traditional IR, there is a wide spectrum of similar techniques and existing methodology that might be further explored as well (e.g., the aforementioned PRF, which infers relevant documents for feedback rather requiring the user to supply them explicitly).

Concept location or feature location represents another task closely related to bug localization. Generally, concept location or feature location aims to identify the relevant parts of a software system that implement a specific concept or functionality. Thus, it is one of the most common activities in program comprehension. Researchers have used a variety of information retrieval techniques in feature location and concept location as well. Marcus et al. [24] used LSI to find modules related to a given feature in form of a user query. Poshyvanyk et al. [28] used LSI first to rank source code elements based on a given feature or bug reports, and then used a Formal Concept Analysis to cluster the results. In another work, Poshyvanyk et al. [27] formulated the feature location problem as a decision-making problem in the presence of uncertainty. The decision is taken based on the opinions from two experts. The first expert is LSI, which enables users to search static documents relevant to a feature. The second expert is the Scenario Based Probabilistic ranking, which helps user rank a list of entities, given a feature of interest, by analyzing dynamic traces from the execution of different scenarios. Gay et al. [12] incorporated RF in IR-based concept location. Although bug reports were used as a concept/feature in some of these studies, they were few in number.

X. Conclusion

Locating bugs is important, difficult, and expensive, particularly for large-scale software projects. To address this, natural language information retrieval (IR) techniques are increasingly being used to suggest potential faulty source files given bug reports. While these techniques are very scalable, in practice their effectiveness remains low in accurately localizing bugs to a small number of files.

Our key insight is that structured IR-based on code constructs, such as class and method names, enables more accurate bug localization. We present BLUiR, which embodies this insight, builds on an open source IR toolkit [40], requires only the source code and bug reports, and takes advantage of bug similarity data if available. We evaluate BLUiR on four open source projects with approximately 3,400 bugs. When bug similarity data is not used, the off-the-shelf IR tookit (unmodified) already exceeds state-of-the-art tool, BugLocator's accuracy. With our enhancements (e.g., structural modeling and camel case indexing), accuracy is significantly improved further. Modeling additional bug similarity data provides yet a further gain. Finally, even if BugLocator is given bug similarity data and BLUiR is not, BLUiR still outperforms BugLocator on three of the four code repositories in the benchmark.

Beyond our technical contributions, our presentation also strives to forge stronger conceptual ties between ongoing work in bug localization and proven practices from the IR community, via a thorough discussion of IR-based bug localization research in relation to fundamental IR theoretical and empirical knowledge and practice.

In our future research, we would like to explore the following areas to further improve our model: bug report summarization and learning parameters.

Bug Report Summarization. In this paper, we showed how the performance of bug localization improves by focusing on condensed information such as bug summaries, class names, or method names. However, we still used exactly the same long bug descriptions from bug reports. There are some automatic techniques [22] that can condense bug descriptions up to 30% of its original size. Such summarized bug descriptions may further improve the performance of bug localization.

Learning to Rank. To tune the value of k_1 and b in our model, we ran BLUiR on AspectJ using a range of values at a fixed interval length and took the pair for which we got the best result for other subject systems. However, the best values may be different for different subject systems. Finding a globally optimal weights is still an open problem in IR research community. Recent work [20] in IR is using machine learning methods to automatically optimize ranking parameters for more sophisticated ranking functions. This would provide another interesting direction for future studies.

We also plan to utilize other datasets (e.g., moreBugs [30]) and perform function/method level bug localization [9].

Acknowledgment: We thank Dawn Lawrie for her helpful feedback. This research was supported in part by NSF Grant SHF-1117902, SRS-0820251, CCF-0845628, and Temple Fellowship.

REFERENCES

[1] R. Abreu, P. Zoeteweij, R. Golsteijn, and A. J. C. Van Gemund. A practical evaluation of spectrum-based fault localization. *J. Syst. Softw.*, 82(11):1780–1792, Nov. 2009.

[2] D. Binkley and D. Lawrie. Information retrieval applications in software maintenance and evolution. *Encyclopedia of Software Engineering*, 2010.

[3] D. M. Blei, A. Y. Ng, and M. I. Jordan. Latent dirichlet allocation. *The Journal of Machine Learning Research*, 3:993–1022, 2003.

[4] S. Brin and L. Page. The anatomy of a large-scale hypertextual web search engine. In *Proceedings of the 7th International Conference on the World Wide Web (WWW)*, pages 107–117, 1998.

[5] F. P. Brooks, Jr. No silver bullet essence and accidents of software engineering. *Computer*, 20(4):10–19, Apr. 1987.

[6] S. Davies, M. Roper, and M. Wood. Using bug report similarity to enhance bug localisation. In *Proceedings of the 2012 19th Working Conference on Reverse Engineering*, WCRE '12, pages 125–134, Washington, DC, USA, 2012. IEEE Computer Society.

[7] F. Diaz. Regularizing query-based retrieval scores. *Information Retrieval*, 10(6):531–562, 2007.

[8] B. Dit, L. Guerrouj, D. Poshyvanyk, and G. Antoniol. Can better identifier splitting techniques help feature location? In *Proceedings of the 2011 IEEE 19th International Conference on Program Comprehension*, ICPC '11, pages 11–20, Washington, DC, USA, 2011. IEEE Computer Society.

[9] B. Dit, M. Revelle, M. Gethers, and D. Poshyvanyk. Feature location in source code: A taxonomy and survey. *Journal of Software: Evolution and Process*, 25(1):53–95, 2013.

[10] E. Enslen, E. Hill, L. Pollock, and K. Vijay-Shanker. Mining source code to automatically split identifiers for software analysis. In *Proceedings of the 2009 6th IEEE International Working Conference on Mining Software Repositories*, MSR '09, pages 71–80, Washington, DC, USA, 2009. IEEE Computer Society.

[11] H. Fang, T. Tao, and C. Zhai. A formal study of information retrieval heuristics. In *Proc. of the ACM SIGIR conference*, pages 49–56, 2004.

[12] G. Gay, S. Haiduc, A. Marcus, and T. Menzies. On the use of relevance feedback in ir-based concept location. In *Proceedings of the IEEE International Conference on Software Maintenance, 2009*, pages 351–360, 2009.

[13] E. Hill, S. Rao, and A. Kak. On the use of stemming for concern location and bug localization in java. In *Proceedings of the12th IEEE International Working Conference on Source Code Analysis and Manipulation*, SCAM '12, 2012.

[14] T. Hofmann. Probabilistic latent semantic indexing. In *Proceedings of the 22nd annual international ACM SIGIR Conference on Research and Development in Information Retrieval*, pages 50–57. ACM, 1999.

[15] D. Hovemeyer and W. Pugh. Finding bugs is easy. *SIGPLAN Not.*, 39(12):92–106, Dec. 2004.

[16] J. Jeon, W. B. Croft, and J. H. Lee. Finding similar questions in large question and answer archives. In *Proc. of the 14th ACM conference on Information and knowledge management*, pages 84–90, 2005.

[17] J. A. Jones and M. J. Harrold. Empirical evaluation of the tarantula automatic fault-localization technique. In *Proceedings of the 20th IEEE/ACM International Conference on Automated Software Engineering*, ASE '05, pages 273–282, New York, NY, USA, 2005. ACM.

[18] M. Lease, J. Allan, and W. B. Croft. Regression Rank: Learning to Meet the Opportunity of Descriptive Queries. In *Proceedings of the European Conference on Information Retrieval*, pages 90–101, 2009.

[19] B. Liblit, M. Naik, A. X. Zheng, A. Aiken, and M. I. Jordan. Scalable statistical bug isolation. In *Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '05, pages 15–26, New York, NY, USA, 2005. ACM.

[20] T.-Y. Liu, J. Xu, T. Qin, W. Xiong, and H. Li. Letor: Benchmark dataset for research on learning to rank for information retrieval. In *Proceedings of SIGIR 2007 Workshop on Learning to Rank for Information Retrieval*, pages 3–10, 2007.

[21] S. K. Lukins, N. A. Kraft, and L. H. Etzkorn. Bug localization using latent dirichlet allocation. *Information and Software Technology*, 52(9):972 – 990, 2010.

[22] S. Mani, R. Catherine, V. S. Sinha, and A. Dubey. Ausum: approach for unsupervised bug report summarization. In *Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering*, FSE '12, pages 1–11, 2012.

[23] C. D. Manning, P. Raghavan, and H. Schütze. *Introduction to Information Retrieval*. Cambridge University Press, 2008.

[24] A. Marcus, A. Sergeyev, V. Rajlich, and J. I. Maletic. An information retrieval approach to concept location in source code. In *Proceedings of the 11th Working Conference on Reverse Engineering*, WCRE '04, pages 214–223, Washington, DC, USA, 2004. IEEE Computer Society.

[25] A. T. Nguyen, T. T. Nguyen, J. Al-Kofahi, H. V. Nguyen, and T. Nguyen. A topic-based approach for narrowing the search space of buggy files from a bug report. In *Proceedings of the 26th IEEE/ACM International Conference on Automated Software Engineering*, pages 263–272, 2011.

[26] J. M. Ponte and W. B. Croft. A language modeling approach to information retrieval. In *Proceedings of the 21st annual ACM SIGIR conference*, pages 275–281, 1998.

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

[28] D. Poshyvanyk and A. Marcus. Combining formal concept analysis with information retrieval for concept location in source code. In *Proceedings of the 15th IEEE International Conference on Program Comprehension*, ICPC '07, pages 37–48, 2007.

[29] S. Rao and A. Kak. Retrieval from software libraries for bug localization: a comparative study of generic and composite text models. In *Proceedings of the 8th Working Conference on Mining Software Repositories*, MSR'11, pages 43–52, 2011.

[30] S. Rao and A. Kak. moreBugs: A New Dataset for Benchmarking Algorithms for Information Retrieval from Software Repositories (tr-ece-13-07). Technical report, Purdue University, School of Electrical and Computer Engineering, April 2013.

[31] S. Robertson, H. Zaragoza, and M. Taylor. Simple bm25 extension to multiple weighted fields. In *Proc. of the 13th ACM Conference on Information and Knowledge Management (CIKM)*, pages 42–49, 2004.

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

[33] J. ROCCHIO. Relevance feedback in information retrieval. *SMART Retrieval System: Experiments in Automatic Document Processing*, 1971.

[34] G. Salton and C. Buckley. Term-weighting approaches in automatic text retrieval. *Information Processing & Management*, 24(5):513–523, 1988.

[35] N. Shahmehri, M. Kamkar, and P. Fritzson. Semi-automatic bug localization in software maintenance. In *Proceedings of the International Conference on Software Maintenance*, pages 30–36, 1990.

[36] E. Y. Shapiro. *Algorithmic Program DeBugging*. MIT Press, Cambridge, MA, USA, 1983.

[37] A. Singhal. Modern information retrieval: A brief overview. *IEEE Data Engineering Bulletin*, 24(4):35–43, 2001.

[38] A. Singhal, C. Buckley, and M. Mitra. Pivoted document length normalization. In *Proceedings of the 19th annual international ACM SIGIR Conference on Research and Development in Information Retrieval*, pages 21–29. ACM, 1996.

[39] B. Sisman and A. Kak. Incorporating version histories in information retrieval based bug localization. In *Proceedings of the 9th IEEE Working Conference on Mining Software Repositories*, pages 50–59, 2012.

[40] T. Strohman, D. Metzler, H. Turtle, and W. B. Croft. Indri: A language model-based search engine for complex queries. In *Proceedings of the International Conference on Intelligent Analysis*, pages 2–6, 2005.

[41] H. K. Wright, M. Kim, and D. E. Perry. Validity concerns in software engineering research. In *Proc. of the FSE/SDP Workshop on Future of Software Engineering Eesearch*, FoSER '10, pages 411–414. ACM, 2010.

[42] X. Yi and J. Allan. A comparative study of utilizing topic models for information retrieval. In *Proceedings of the 31st European Conference on Information Retrieval (ECIR)*, pages 29–41. Springer-Verlag, 2009.

[43] A. Zeller and R. Hildebrandt. Simplifying and isolating failure-inducing input. *IEEE Trans. Softw. Eng.*, 28(2):183–200, Feb. 2002.

[44] C. Zhai. Notes on the lemur tfidf model (unpublished work). Technical report, Carnegie Mellon University, 2001.

[45] X. Zhang, H. He, N. Gupta, and R. Gupta. Experimental evaluation of using dynamic slices for fault location. In *Proceedings of the 6th International Symposium on Automated Analysis-driven Debugging*, AADEBUG'05, pages 33–42, New York, NY, USA, 2005. ACM.

[46] 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 *Proceedings of the 2012 International Conference on Software Engineering*, ICSE 2012, pages 14–24, Piscataway, NJ, USA, 2012. IEEE Press.