2013

Exploiting Spatial Code Proximity and Order for Improved Source Code Retrieval for Bug Localization

Bunyamin Sisman
Purdue University, bsisman@purdue.edu

Avinash Kak
School of Electrical and Computer Engineering, Purdue University, West Lafayette, IN campus, kak@purdue.edu

Exploiting Spatial Code Proximity and Order for Improved Source Code Retrieval for Bug Localization

Bunyamin Sisman, Avinash C. Kak

Abstract—Practically all Information Retrieval (IR) based approaches developed to date for automatic bug localization are based on the bag-of-words assumption that ignores any positional and ordering relationships between the terms in a query. In this paper we argue that bug reports are ill-served by this assumption since such reports frequently contain various types of structural information whose terms must obey certain positional and ordering constraints. It therefore stands to reason that the quality of retrieval for bug localization would improve if these constraints could be taken into account when searching for the most relevant files. In this paper, we demonstrate that such is indeed the case. We show how the well-known Markov Random Field (MRF) based retrieval framework can be used for taking into account the term-term proximity and ordering relationships in a query vis-a-vis the same relationships in the files of a source-code library to greatly improve the quality of retrieval of the most relevant source files. We have carried out our experimental evaluations on popular large software projects using over 4 thousand bug reports. The results we present demonstrate unequivocally that the new proposed approach is far superior to the widely used bag-of-words based approaches.

Index Terms—Term Proximity, Term Dependence, Information Retrieval, Markov Random Fields, Bug Localization

1 Introduction

Code search plays an important role in software development and maintenance. The tools that are deployed today for code search range all the way from simple command-line functions like 'grep' to complex search facilities tailored for the specific needs of the developers. These different types of search facilities are used to locate various parts of a software library for concept location, change impact analysis, traceability link analysis, and so on [1], [2], [3], [4], [5], [6]. Our particular interest lies in a class of code search tools that are based on Information Retrieval (IR) techniques and our end goal is bug localization. For automatic bug localization, we treat the bug reports as text queries and the corresponding software constructs that should be modified to implement a fix for the bugs as the relevant artifacts that should be retrieved by the search tool [7], [8], [9].

Obviously, the success of an IR framework that leverages bug reports for automatic bug localization depends much on how the bug reports are represented vis-a-vis the source code files and other documents in a library. In the widely used *bag-of-words* representations for both the queries and the source code documents, all positional and ordering relationships between the terms are lost [10], [7], [9]. To us, this is tantamount to a serious loss of information considering that a bug report, in general, is a composition of structured and unstructured textual data that frequently includes (a) patches; (b) stack traces when the software fault throws an exception; (c) snippets of code; (d) natural language sentences; and so on [11], [8]. Patches and stack traces, especially, contain vital inter-term proximity and ordering relationships that ought to be exploited for the purpose of retrieval. Say, if two terms are proximal to each other in a stack trace, you'd want a source code file containing similar code to have the same two terms in a similar proximal relationship. The work we report in this paper demonstrates that the quality of retrieval improves greatly when a retrieval framework allows for ordering and positional (through proximity) relationships to be taken into account in the retrieval process.

As to how to incorporate ordering and positional relationships in a retrieval framework, we have several possibilities at our disposal that have been examined in the past mostly in the context of retrieval from natural language corpora. At one end of the spectrum, we have *ad hoc* approaches such as those that compute term co-occurrence frequencies and proximity-based term-term dependencies. And, at the other end of the spectrum, we have more principled approaches, such as those based on Markov Random Fields (MRF) [12] that are based on modeling the term-term dependencies by graphs whose arcs capture the inter-term relationships in the queries vis-a-vis the same in the documents.

With regard to the investigation of such approaches in retrieval from software libraries, in a recent contribution [13], we presented an *ad hoc* approach that demonstrated how the inter-term proximities can be used to reformulate the queries in order to improve the quality of retrievals. The query reformulation in that approach takes place through a two-step process that is carried

• *Authors are with the Department of Electrical and Computer Engineering, Purdue University, West Lafayette, IN, 47907.*
E-mail: bsisman,kak@purdue.edu

out without any additional input from the user. In the first step, the top-ranked retrievals for the user-supplied query are analyzed for the detection of terms that are proximal to the query terms as supplied by the user. The proximal terms thus discovered are used to enrich the original query in the second step. We showed that the retrieval for the query reformulated in this way is significantly more precise than it is for the original query.

We now show that even more significant improvement in retrieval precision can be obtained by using a more principled alternative to *ad hoc* approaches. In particular, we will show that when an MRF is used to model the ordering and the positional dependencies between the query terms vis-a-vis the documents, we end up with a framework that not only yields a higher retrieval precision with the simplest of the ordering and proximity constraints, but that can also be generalized to the investigation of more general such constraints. In the MRF based approach, certain subsets of the terms in a bug report are used for scoring the software artifacts while taking into account term-term proximity and order. This approach exploits the fact that the software artifacts that contain the query terms in the same order and/or in similar proximities as in the query itself are more likely to be relevant to a given query.

While, as demonstrated by our results, MRF is a powerful approach to the modeling of query-document relationships, to fully exploit its potential it must be used in conjunction with what we refer to as *Query Conditioning* (QC). The goal of QC is to recognize the fact that a bug report constitutes a highly structured query whose various parts consist, as we mentioned previously, of a textual narrative, a stack trace, etc [11]. These different parts are disparate in the sense that the inter-term relationships do not carry the same weight in them. For example, the proximity of the terms used in the stack trace portions of a bug report carries far more weight than in the textual narrative. Therefore, the ordering and proximity constraints are likely to be far more discriminative in those portions of bug report that, by their very nature, are far more structured. To further underscore the importance of these structured portions of the bug reports, past studies have shown that stack traces are one of the most valuable source of information to pinpoint the location of the bugs [14], [15].

We have named the overall code retrieval engine that includes both QC and MRF modeling as Terrier+. The reason for the '+' suffix in the name is that ours is an enhancement of the popular open-source research information retrieval engine called Terrier[1]. The functionality we have added to create Terrier+ is highly modular, in the sense that any of enhancements can be turned on and off merely by clicking buttons in the GUI in order to measure the retrieval effectiveness of that enhancement. This allows for a convenient assessment of the retrieval power that can be attributed to each enhancement.

1. http://terrier.org

Note that whereas Terrier+ applies MRF modeling to all queries, QC becomes an important factor only when structured elements are present in the queries. In general, detecting the structured elements in bug reports is a difficult task as they may have different formats and they are usually surrounded by other types of textual data [11]. It is also not uncommon for these constructs to undergo unexpected format changes, such as those caused by accidental line breaks, when they are copied into a bug report. In order to overcome these challenges, Terrier+ employs several regular expressions to detect and extract these structured elements from bug reports.

We experimentally validate the proposed bug localization framework on three large software libraries: AspectJ, Google Chrome, and Eclipse 3.1. We show that MRF modeling of the queries and the query conditioning step (whenever the queries lend themselves to such conditioning) significantly improve the accuracy with with which the bugs can be localized. In order to investigate the effect of the length of queries on the precision with which the bugs are localized, we carried out retrievals with just the bug report titles and with the bug reports taken in their entirety. Whereas MRF modeling resulted in improved precision in bug localization even for short queries consisting of just the bug report titles, the improvements were even more significant when the bug reports in their entirely were subject to MRF modeling and the QC step. Our experimental results also include comparison with the other state of the art IR based approaches to bug localization. We demonstrate that, on the average, the MRF and QC based framework outperforms all these other approaches.

This paper is organized as follows. In the next section, we start by stating the research questions addressed in the experimental validation of the proposed code retrieval framework. Seeing these questions at the outset will hopefully give the reader a better sense of the scope of our work, especially in relation to the previous related contributions by us and by others. We then present the proposed MRF modeling along with Query Conditioning. We evaluate our retrieval framework in Section 4. The relevant work is presented in Section 5. Section 6 provides the possible threats to the validity of the proposed approach. Finally, we conclude in Section 7.

2 Research Questions

Our empirical evaluation on large open-source software projects shows that our proposed retrieval framework leads to significant improvements in automatic bug localization accuracy. As mentioned in the previous section, our retrieval engine has been packaged as an enhancement to the popular research IR retrieval engine known as Terrier and we refer to our enhancement as Terrier+.

We compare the performance of Terrier+ to the other IR approaches developed for the retrieval of the buggy source files in response to bug reports. Included in these other IR-based approaches is the Spatial Code Proximity (SCP) based Query Reformulation (QR) algorithm in which a given short query is enriched with additional informative terms drawn from the highest ranked retrieval results with respect to the original query for an improved retrieval accuracy [13]. Another important class of IR tools developed for automatic bug localization leverages the past development efforts. Such IR tools have been also shown to improve the bug localization accuracy significantly [9], [7].

For the evaluation of Terrier+, we conducted extensive validation tests with the following research questions at hand:

RQ1: *Does including code proximity and order improve the retrieval accuracy for bug localization. If so, to what extend?*

RQ2: *Is QC effective on improving the query representation vis-a-vis the source code?*

RQ3: *Does including stack traces in the bug reports improve the accuracy of the bug localization?*

RQ4: *How does the MRF-based retrieval framework compare with the Query Refomulation (QR) based retrieval frameworks for bug localization?*

RQ5: *How does the MRF-based retrieval framework compare to the other bug localization frameworks that leverage the past development history?*

3 The Proposed Approach

Directly or indirectly, all traditional IR based approaches to bug localization amount to comparing the first-order distribution of terms in a query vis-a-vis the documents. This applies as much to simple approaches based on VSM and Unigrams [10], [7] as it does to approaches based on LDA [2], [10], [16] that use hidden variables for injecting additional degrees of freedom for comparing the queries with the documents. All of these approaches miss out on the information contained in the inter-term relationships in the queries and in the documents.

While we do now know how to extend the bag-of-words approaches of the sort mentioned above with proximity-based reformulation of a query for improving the quality of retrievals [13], what we need are theoretically well-grounded approaches that can be generalized to the incorporation of arbitrary inter-term relationships between the terms of a query vis-a-vis the documents. The goal of this section is to address this need by using the notion of Markov Random Fields (MRF).

In the subsections to follow, we first review how the inter-term dependencies are modeled with MRF. We then mention three different specializations of MRF modeling that appear to be particularly appropriate for our needs. Subsequently, in order to exploit MRF modeling to the maximum, we present our Query Conditioning (QC) method to extract from a query those portions that are particularly suited to modeling by MRF. Note that, as we will demonstrate later, MRF improves retrievals even in the absence of QC. However, by giving greater weight to the inter-term relationships in those portions of a query that QC has identified to be as being highly structured, we improve the quality of retrievals much further.

3.1 Markov Random Fields

Over the years, researchers in the machine learning community have devoted much energy to the investigation of methods for the probabilistic modeling of arbitrary dependencies amongst a collection of variables. The methods that have been developed are all based on graphs. The nodes of such graphs represent the variables and the arcs the pairwise dependencies between the variables. The graphs may either be directed, as in Bayesian Belief Networks [17], or undirected, as in the networks derived from Markov Random Fields [12], [17]. In both these methods, the set of variables that any given variable directly depends on is determined by the node connectivity patterns. In a Bayesian Belief Network, the probability distribution at a node q is conditioned on only those nodes that are at the tail ends of the arcs incident on q, taking the *causality* into account. In a Markov Network, on the other hand, the probability distribution at a node q depends on the nodes that are immediate neighbors of q without considering any directionality. In the context of retrieval from natural language corpora, the work of Metzler and Croft [12] has shown that Markov Networks are particularly appropriate for the modeling of inter-term dependencies vis-a-vis the documents.

In general, given a graph G whose arcs express pairwise dependencies between the variables, MRF modeling of the probabilistic dependencies amongst a collection A of variables is based on the assumption that the joint distribution over all the variables in the collection can be expressed as product of non-negative potential functions over the cliques in the graph:

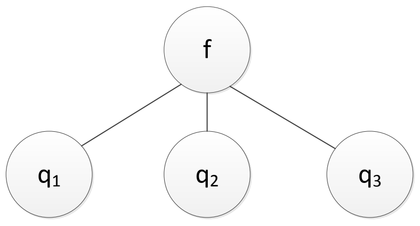
$$P(A) \quad = \quad \frac{1}{Z}\prod_{k=1}^{K}\phi(C_k) \tag{1}$$

where $\{C_1, C_2, \ldots, C_K\}$ represents the set of all cliques in the graph G, and $\phi(C_k)$ a non-negative potential function associated with the clique C_k. In the expression above, Z is merely for the purpose of normalization since we want the sum of $P(A)$ over all possible values that can be taken by the variables in A to add up to unity.

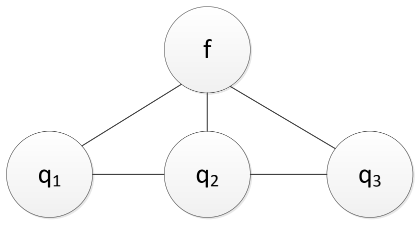
Our end goal with MRF is to rank the files in the code base according to the probability of a file f in the software library to be relevant to a given query Q. We denote this probability by $P(f|Q)$ [7]. Using the definition of the conditional probability, we can write

$$P(f|Q) = \frac{P(Q,f)}{P(Q)}. \tag{2}$$

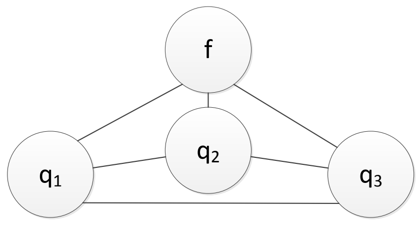
As we are only interested in ranking the files and the denominator in Eq. 2 does not depend on files, it can be ignored. Hence $P(f|Q) \overset{rank}{=} P(Q,f)$. In order to separate



(a) Full Independence



(b) Sequential Dependence



(c) Full Dependence

Fig. 1. *To illustrate the notion of Markov network for capturing the inter-term dependencies in a query vis-a-vis a document, the case depicted is based on a query consisting of just three terms. We show three different MRF based models.*

out the roles played by the variables that stand for the query terms (since we are interested in the inter-term dependencies in the queries) vis-a-vis the contents of a source file f, as suggested by Metzler and Croft [12], we will use the following variation of the general form expressed in Eq. 1 to compute this joint probability:

$$P(Q,f) \;=\; \frac{1}{Z}\prod_{k=1}^{K}\phi(C_k) \;\overset{rank}{=}\; \sum_{k=1}^{K} log(\phi(C_k)) \qquad (3)$$

where Q stands for a query which is assumed to consist of the terms $q_1, q_2, ..., q_{|Q|}$ and f a file in the software library. The nodes of the graph G in this case consist of the query terms, with one node for each term. G also contains a node that is reserved for the file f whose relevancy to the query is in question. As before, we assume that this graph contains the cliques $\{C_1, C_2, \ldots, C_K\}$. As shown in the formula, for computational ease it is traditional to express the potential $\psi(C_k)$ through its logarithmic form, that is through $\psi(C_k) = log(\phi(C_k))$.

The fact that a fundamental property of any Markov network is that probability distribution at any node q is a function of only the nodes that are directly connected to q may now be expressed as

$$P(q_i|f, q_{j\neq i} \in Q) = P(q_i|f, q_j \in neig(q_i)) \qquad (4)$$

where $neig(q_i)$ denotes the terms whose nodes are directly connected to the node for q_i. As observed in [12], this fact allows arbitrary inter-term relationships to be encoded through appropriate arc connections amongst the nodes that represent the query terms in the graph G. At one end of the spectrum, we can assume that the query terms are all independent of one another by the absence of any arcs between them. This assumption, known as the usual bag-of-words assumption in information retrieval, is referred to as the *Full Independence (FI)*. And at the other end of the spectrum, we may assume a fully connected graph in which the probability distribution at each node representing a query term depends on all the other query terms (besides being dependent on the file f). This is referred to as the *Full Dependence (FD)*. Fig. 1a and 1c depict the graph G for FI and FD assumptions for the case when a query Q consists of exactly three terms.

What makes MRF modeling particularly elegant is that it gives us a framework to conceptualize any number of other "intermediate" forms of dependencies that are between the two extremes of the FI and the FD assumptions. This we can do by simply choosing graphs G of different connectivity patterns. Whereas FI is based on the absence of any inter-term arcs in G and FD on there being an arc between each query term and every other term, we may now think of more specialized dependencies such as the one depicted in Fig. 1b. This dependency model, referred to as the *Sequential Dependency (SD)* model in [12], incorporates both order and proximity between a sequence $(q_1, q_2, \ldots, q_{|Q|})$ of query terms.

At this point, the reader may wonder as to how one would know in advance as to which connectivity pattern to use for the graph G. The connectivity pattern is obviously induced by the software library itself. Suppose a phrase level analysis of the files in the library indicates that the phrase "interrupt sig handler" occurs in the files and *can be used to discriminate between them*, you would want the nodes for the terms "interrupt," "sig," and "handler" to be connected in the manner shown in Fig. 1b. This is because the SD model shown in that figure would only allow for pairwise (but ordered) occurrences of the words "interrupt," "sig," and "handler" to be matched in the files. The frequencies with which these ordered terms appear in the files may also carry discriminatory power. The relative importance of the words occurring individually or in ordered pairs would be determined by their relative frequencies in the files. In contrast to the case depicted in Fig. 1b, should it happen that the queries and the relevant files contain the three terms "interrupt," "sig," and "handler" in all possible orders, you would want to use the FD assumption depicted in Fig. 1c. In this case, the number of times these terms occur together within a window of a certain size would carry discriminatory power for choosing the files relevant to a query. Finally, should it happen, that the three terms occur in the relevant files without there being a phrasal sense to their appearance in the files, you would want to use the bag-of-words, FI, assumption.

We are particularly interested in the graph connectivity induced by the notion of Spatial Code Proximity

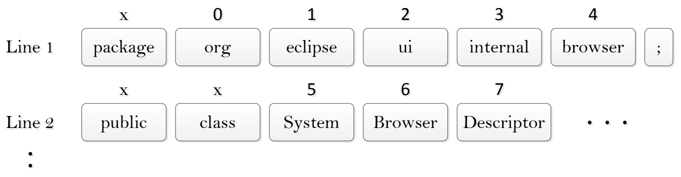


Fig. 2. *An illustration of indexing the positions of the terms in an Eclipse Class: SystemBrowerDescriptor.java. The 'x' symbol indicates the stop-words that are dropped from the index.*

(SCP) [13]. SCP consist of first associating a positional index with each term in a query and in the documents as shown in Fig. 2. Our goal is to translate the values of the positional indexes into graph models based on the FI, SD, and FD assumptions. In the next three subsections, we will present formulas that show how these models can be derived from SCP based indexes.

3.1.1 Full Independence (FI)

As already stated, the FI assumption reduces an MRF model to the usual bag-of-words model that has now been extensively investigated for automatic bug localization [10], [7], [9]. As should be clear from the graph representation of this model depicted in Fig. 1a for the case of a query with exactly three terms, FI modeling involves only 2-node cliques. Therefore, under MRF modeling, the probability of a query given a file is simply computed by summing over the 2-node cliques: $P_{FI}(f|Q) \overset{rank}{=} \sum_{i=1}^{|Q|} \psi_{FI}(q_i, f)$. The choice of the potential function, obviously critical in computing this probability, should be in accord with the fact that MRF under FI assumption amount to the bag-of-words modeling. Therefore, a good choice is to make the potential $\psi_{FI}(q_i, f)$ proportional to the frequency of the query term q_i in the file f. Since the zero probability associating with a query term q_i that does not appear in a file f can create problems when estimating the relevance of f to a query, it is common to add what is referred to as a smoothing increment to the term frequencies. A powerful smoothing approach, known as Dirichlet smoothing, uses the frequency of a term in the entire corpus [7]. Shown below is a formula for the potential $\psi_{FI}(q_i, f)$ that includes Dirichlet smoothing:

$$\psi_{FI}(q_i, f) \ = \ \lambda_{FI} log\Big(\frac{tf(q_i, f) + \mu P(q_i|C)}{|f| + \mu}\Big) \qquad (5)$$

where $P(q_i|C)$ denotes the probability of the term in the whole collection, $tf(q_i, f)$ is the term frequency of q_i in a file f, $|f|$ denotes the length of the file and μ is the Dirichlet smoothing parameter. The model constant λ_{FI} has no impact on the rankings with this model. However, we keep it in the formulation as we will use it later in SD and FD modeling.

The probability expression shown above for the relevance of a term to a file is exactly the same as it appears in the widely used bag-of-words model known as the *Smoothed Unigram Model (SUM)* whose usefulness in automatic bug localization has been demonstrated in [10], [7], [9]. We will use the retrieval results obtained with FI as the baseline in order to determine the extent of improvements one can obtain with the other two models, SD and FD.

3.1.2 Sequential Dependence (SD)

The SD model takes the order and the proximity of the terms into account in such a way that the probability law for a query term q_i given a file f obeys $P(q_i|f, q_j \in \{q_1, \ldots, q_{i-1}, q_{i+1}, \ldots, q_{|Q|}\}) = P(q_i|f, q_{i-1}, q_{i+1})$.

To see how a software library can be processed to induce the SD model, note from the example shown in Fig. 1b that we now have 3-node cliques in addition to the 2-node cliques of the FI model. Therefore, we must now count the frequencies with which pairs of terms occur together, with one following the other (without necessarily being adjacent) in a specific order, in addition to counting the frequencies for the terms occurring singly as in the FI model [18]. Again incorporating Dirichlet smoothing for the same reasons as in the FI model, we employ the following potential function for the 3-node cliques corresponding to a file f and two consecutive query terms q_{i-1} and q_i:

$$\psi_{SD}(q_{i-1}, q_i, f) = \lambda_{SD} \; log\Big(\frac{tf_W(q_{i-1}q_i, f) + \mu P(q_{i-1}q_i|C)}{|f| + \mu}\Big) \qquad (6)$$

where $tf_W(q_{i-1}q_i, f)$ is the number of times the terms q_{i-1} and q_i appear in the same *order* as in the query within a window length of $W \geq 2$ in the file. For $W > 2$, the terms do not have to be adjacent in the file and the windows may also contain other query terms. The smoothing increment $P(q_{i-1}q_i|C)$ is the probability associated with the pair $(q_{i-1}q_i)$ in the entire software library. To the potential function shown above, we must now add the potential function for 2-node cliques the reader has already seen for the FI model:

$$P_{SD}(f|Q) \overset{rank}{=} \sum_{i=2}^{|Q|} \psi_{SD}(q_{i-1}, q_i, f) + \sum_{i=1}^{|Q|} \psi_{FI}(q_i, f). \quad (7)$$

As the reader would expect, the ranking of the files with the potential function shown in Eq. 7 is only sensitive to the relative weights expressed by the model parameters λ_{FI} and λ_{SD}, the overall scaling of these weights being inconsequential on account of the unit summation constraints on probabilities. We therefore set $\lambda_{FI} + \lambda_{SD} = 1$. We can think of λ_{SD} as an interpolation or a mixture parameter that controls the relative importance of the 3-node cliques vis-a-vis the 2-node cliques.

3.1.3 Full Dependence (FD)

As demonstrated previously by Fig. 1c, the FD assumption implies a fully connected graph G whose nodes correspond to the individual query terms, with one node being reserved for the file f under consideration. The graph being fully connected allows for a file f to be considered relevant to a query regardless of the order in which the query terms occur in the file. (Compare this to the SD case where, for a file f to be considered relevant to a query, it must contain the query terms in the same order as in the query.) Therefore, the FD assumption provides a more flexible matching mechanism for retrievals.

The price to be paid for the generality achieved by FD is the combinatorics of matching all possible ordering of the query terms with the contents of a file. To keep this combinatorial explosion under control, following [18], we again limit ourselves to just 2-node and 3-node cliques. While this may sound the same as for the SD assumption, note that the 3-node cliques are now allowed for a pair of query terms for both ordering of the terms. Therefore, for any two terms q_i and q_j of the query, the potential function takes the following form for the 3-node cliques:

$$\psi_{FD}(q_i, q_j, f) = \lambda_{FD} log\Big(\frac{tf_W(q_i q_j, f) + \mu P(q_i q_j|C)}{|f| + \mu}\Big) \tag{8}$$

where λ_{FD} again works as a mixture parameter similar to λ_{SD}, i.e. $\lambda_{FI} + \lambda_{FD} = 1$; μ is the smoothing parameter and $tf_W(q_i q_j, f)$ is the frequency for the pair $q_i q_j$ in f. Summing over the cliques, we obtain the ranking score of a file by

$$P_{FD}(f|Q) \overset{rank}{=} \sum_{i=1}^{|Q|} \sum_{j=1, j\neq i}^{|Q|} \psi_{FD}(q_i, q_j, f) + \sum_{i=1}^{|Q|} \psi_{FI}(q_i, f). \tag{9}$$

3.1.4 A Motivating Example

We will now use a simple example to compare the retrieval effectiveness of the three models, FI, SD, and FD. For reasons of space limitations, we will limit our example to the simplest of the bug reports, one that only contains a one-line text narrative which corresponds to the title of the bug report.

The bug 98995[2] filed for Eclipse v3.1 has a title that reads: *"Monitor Memory Dialog needs to accept empty expression"*. The target source files that were eventually modified to fix this bug are:

1) org.eclipse.debug.ui/.../ui/views/memory/MonitorMemoryBlockDialog.java
2) org.eclipse.debug.ui/.../ui/views/memory/AddMemoryBlockAction.java
3) org.eclipse.debug.ui/.../ui/DebugUIMessages.java.

2. https://bugs.eclipse.org/bugs/show_bug.cgi?id=98995

TABLE 1
Retrieval accuracies for the Bug 98995 with three different MRF models.

Method	2-node cliques	3-node cliques	Ranks	AP
FD	7	42	1-3-2	1.0000
SD	7	6	2-3-4	0.6389
FI	7	0	6-5-10	0.2778

After removing the stop-words from the title, the final query consists of seven unique terms. Table 1 presents the retrieval accuracies obtained for this query, along with the number of cliques utilized for each dependency assumption. In the table, the column "Rank" gives the ranks of the three relevant files in the ranked lists retrieved. AP is the resulting Average Precision.

Investigating the ranked lists returned for the three models, we see that FI ranks several irrelevant files above the relevant ones. One such file is ASTFlattener.java. Although this file does not contain any of the terms "monitor", "memory" and "dialog"; it is retrieved at the top rank by this model because, as a file related to parsing the *Abstract Syntax Trees (AST)*, it contains the terms "accept", "empty" and "expression" with very high frequencies. Clearly, the model misses the context of the query.

In comparison to FI, SD is able to retrieve the relevant files at higher ranks, as shown in Table 1. The improvement obtained with SD is a consequence of the discriminations achieved by requiring that the query terms, when they appear together in a source file, do so in a specific order. The creation of the 3-node cliques with the query terms is illustrated in Fig. 3. A 3-node clique is formed by the two words depicted together with an under-bracket and the node corresponding to a file. Since the relevant files contain these term blocks in close proximity with high frequencies; with this model, they receive higher ranking scores in comparison to the irrelevant files.

Despite the improvements, SD still ranks one irrelevant file, ASTRewriteFlattener.java, above all the relevant ones. This file also does not contain any of the terms "monitor", "memory" and "dialog". However, it contains in close proximity the term pairs from the 2 of the 3-node cliques: "accept empty" and "empty expression". The file manages to receive a high ranking score with these term pairs in addition to the AST related terms.

FD captures the context of the query better than the other two models by considering all the term pairs in the query regardless of their position and order. It assumes that any pair of query terms can depend on one another, hence the number of cliques it uses is higher. This modeling approach ranks the three relevant files at the top ranks above any irrelevant files and reaches a perfect average precision of 1.0.

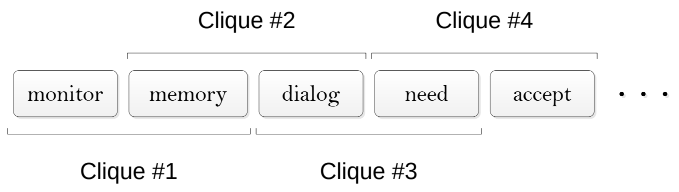


Fig. 3. *Clique creation for the Bug 98995. The figure shows the first 4 query term blocks for the 3-node cliques utilized by the SD modeling.*

3.2 Query Conditioning

When a bug report contains highly structured components, such as a stack trace and/or a source code patch [11], such information can be crucial to locating the files relevant to the bug [15]. Being highly structured, these components must first be identified as such and subsequently processed to yield the terms that can then be used to form a query for IR based retrieval. The processing steps needed for that purpose will be different for the two different types of components we consider: stack traces and source code patches. We refer to the collection of these steps as *Query Conditioning (QC)*. QC is carried out with a set of regular expressions that, while custom designed for the different types of structured components encountered, are sufficiently flexible to accommodate small variations in the structures.[3]

As already stated, our retrieval framework has been packaged as an enhancement to the popular research IR retrieval engine Terrier and we refer to our enhancement as Terrier+. With regard to the flow of processing related to QC in Terrier+, it uses regular expressions to first identify the patches and the stack traces from a given bug report if any of these elements are available in the report. Then, it processes them separately to sift out the most relevant source code identifiers to be used in the retrievals. The final query is composed from the terms extracted from the stack traces and the patches if one or both of these components are available. If these structured components are not available, Terrier+ makes do with the entire bug report such as it is and feeds it into the MRF framework.

3.2.1 Stack Traces

When Terrier+ detects the stack traces in a bug report, it automatically extracts the most likely locations of the bug by identifying the methods in the trace. As the call sequence in a stack trace starts from the most recent method call,[4] we extract only the topmost T methods while discarding the rest of the trace since the methods down in the trace have a very little chance of containing any relevant terms and they are likely to introduce noise into the retrieval process. Fig. 4 illustrates the stack trace that was included in the report for Bug 77190 filed for Eclipse[5]. The bug caused the *EmptyStackException* to be thrown by the code in PushFieldVariable.java and was subsequently fixed in a revised version of this code. The figure highlights the extracted portion of the stack trace that is used in forming the final query. Note that we only extract the methods that are present in the code base to which the bug report applies. That is, we skip the methods from the libraries belonging to the Java platform itself, as illustrated in the figure. During the experiments, we empirically set $T = 3$, as this setting resulted in the best retrieval accuracies on the average. As we show in our experimental evaluation, this filtering approach increases the precision of the retrievals significantly.

3. Our QC only takes into account the stack traces and source code patches when they can be identified in a bug report. Note that a bug report may also contain additional source code snippets that are not meant to be patches [11]. QC treats any additional such code on par with the main textual part of the report.

4. We do realize that for some languages the methods in a stack trace are in the opposite order. That is, the most recent method call appears as the last entry in the trace. The logic of identifying the methods most relevant to a bug would obviously need to be reversed for such languages.

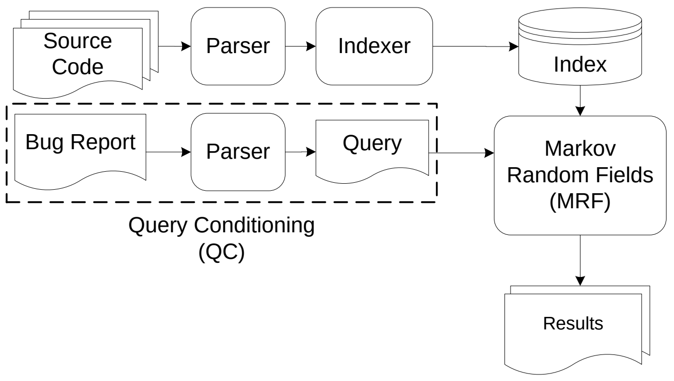


Fig. 5. *An illustration of the data flow in the proposed retrieval framework.*

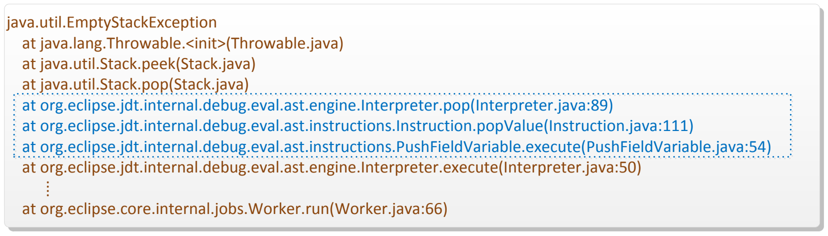
3.2.2 Patches

Source code patches are included in a bug report when a developer wishes to also contribute a possible (and perhaps partial) fix to the bug. When contributed by an experienced developer, these components of a bug report can be directly used for pinpointing the files relevant to a bug.

A patch for a given bug is usually created with the *Unified Format* to indicate the differences between the original and the modified versions of a file in a single construct. With this format, the textual content of the patch contains the lines that would be removed or added in addition to the contextual lines that would remain unchanged in the file after the patch is applied. For term extraction from the patches, Terrier+ does not use the lines that would be added after the suggested patches are applied to the files as those lines are not yet present in the code base.

Obviously, the files mentioned by a developer in a patch may not correspond to the actual location of the

5. https://bugs.eclipse.org/bugs/show_bug.cgi?id=77190



```
java.util.EmptyStackException
   at java.lang.Throwable.<init>(Throwable.java)
   at java.util.Stack.peek(Stack.java)
   at java.util.Stack.pop(Stack.java)
   at org.eclipse.jdt.internal.debug.eval.ast.engine.Interpreter.pop(Interpreter.java:89)
   at org.eclipse.jdt.internal.debug.eval.ast.instructions.Instruction.popValue(Instruction.java:111)
   at org.eclipse.jdt.internal.debug.eval.ast.instructions.PushFieldVariable.execute(PushFieldVariable.java:54)
   at org.eclipse.jdt.internal.debug.eval.ast.engine.Interpreter.execute(Interpreter.java:50)
       ⋮
   at org.eclipse.core.internal.jobs.Worker.run(Worker.java:66)
```

Fig. 4. *The stack trace that was included in the report for Bug 77190 filed for Eclipse. With QC, the trace is first detected in the report with regular expression based processing. Subsequently, the highlighted lines are extracted as the most likely locations of the bug and fed into the MRF framework.*

TABLE 2
Evaluated Projects

Project, Description	Language	$\|B\|$	$\|RF\|$	$\|Q_{Title}\|$
AspectJ, An Extension to Java Programming Language	Java	291	3.09	5.78
Eclipse v3.1, Integrated Development Environment	Java	4,035	2.76	5.80
Chrome v4.0, WEB Browser	C/C++	358	3.82	6.21

bug. And, there may be additional files in the code base that may require modifications in the final fix for the bug. While the importance of information in such source code patches cannot be overstated, it is important to bear in mind that their inclusion in the bug reports is more the exception than the rule. Out of the 4,035 bug reports we analyzed for Eclipse v3.1, only 8 contained a patch. Along the same lines, out of the 291 bug reports we analyzed for the AspectJ project, only 4 contained a patch. Nonetheless, considering the importance of the information contained in the patches when they are included in a bug report, Terrier+ takes advantage of that information whenever it can.

Fig 5 illustrates the data flow in the retrieval framework with QC and MRF.

4 Experimental Evaluation

We evaluate the effect of incorporating term dependencies on the retrievals for bug localization on three large software projects, namely Eclipse IDE[6], AspectJ[7] and Google Chrome[8]. We evaluate Query Conditioning (QC) on only Eclipse and AspectJ as the bug reports for Chrome do not contain stack traces or patches. We use a set of bug reports, denoted B, that were filed for these projects and, for ground truth, the files modified to fix the corresponding bugs as the relevant file set to be retrieved by the retrieval engine. The relevant file set for a bug report is denoted by RF.

Since the bug tracking databases such as Bugzilla[9] do not usually store the modification histories of the changes made to the files in response to the bug reports, researchers commonly use the commit messages in the repository logs in order to link the modifications to the bug reports in a bug tracking database [9], [13], [19], [20]. The *BUGLinks*[10] [13] and the *iBugs* [20] are the resulting datasets of such approaches that reconstruct the links between the bug reports and the files relevant to the bugs in the repositories for the projects we have used. The BUGLinks dataset contains information related to the Eclipse and the Chrome projects, whereas the iBugs dataset contains information related to the AspectJ project. Tables 2 and 3 present various statistics drawn from these datasets regarding the three projects used in our evaluation study. In Table 2, $|B|$ denotes the number of bug reports used in querying the code base of each project, $|RF|$ the average number of relevant files per bug, and $|Q_{Title}|$ the average lengths of the bug report titles that are used in the retrievals.[11] Table 3 presents the statistics of the bug reports used in the evaluation of the MRF framework along with QC. In the table, #Patches and #Stack Traces show the number of bug reports containing patches and stack traces, respectively, and $|Q_{Title+Desc}|$ is the average lengths of the bug reports, including both the title and the description parts without any filtering, in terms of the number of tokens used in querying the code base.

In the rest of this section, we will start out with how the source files and the bug reports are tokenized for the extraction of the terms that are used to represent each file. Subsequently, we first describe the metrics we use in our evaluation study. That is followed by the experimental results demonstrating the power of the MRF based approach to the modeling the source code libraries.

4.1 Preprocessing of the Source Code Files and the Bug Reports

For the indexing of a particular version of the target code base, we first split the compound terms using

6. www.eclipse.org
7. http://eclipse.org/aspectj/
8. www.google.com/chrome
9. https://bugs.eclipse.org/bugs/

10. https://engineering.purdue.edu/RVL/Database/BUGLinks/
11. As we describe in Section 4.3, our experimental studies involve two types of experiments: those that are based on just the titles of the bug reports, and those that include the titles and the bug descriptions. Obviously, whereas MRF modeling can be applied to both types of queries, query conditioning (QC) can only be investigated for the latter type.

TABLE 3
Various statistics related to the bug reports used in the experiments for the evaluation of MRF and QC.

Project	#Patches	#Stack Traces	$\lvert Q_{Title+Desc}\rvert$
AspectJ	4	81	56.77
Eclipse v3.1	8	519	44.11

punctuation characters and camel casing. Then we drop the programming language specific terms and a set of standard English stop words. The remaining terms are then stemmed into their common roots using the Porters stemming algorithm [21]. The position of each term extracted from the files is recorded after these preprocessing steps, as illustrated in Fig. 2. These steps constitute the front-end to the enhancements that distinguish Terrier+ from Terrier. As mentioned previously, Terrier+ is an extension to the open-source research search engine Terrier [22]. As for the bug reports, they are also subject to the same preprocessing steps.

Subsequent to preprocessing, Terrier+ represents a file my a multidimensional array that can be accessed via its ID. This data structure contains term IDs, the corresponding term frequencies and the positions of the terms in the file. For each term in a given query, the files that contain the term are accessed via an *Inverted Index* in which a term is represented by a two dimensional array that stores the file IDs and the frequency of the term in those files.

4.2 Evaluation Metrics

We evaluate the retrieval accuracy of Terrier+ using precision and recall based metrics [21]. We have tabulated the bug localization performance using precision at rank r ($P@r$), recall at rank r ($R@r$) and Mean Average Precision (MAP) metrics. While $P@r$ measures the accuracy on the retrieved set of files, recall evaluates the completeness of the retrievals. The average precision (AP) for a query $Q \in B$, on the other hand, is given by

$$AP(Q) = \frac{\sum_{r=1}^{RT} P@r \times I(r)}{rel_Q} \tag{10}$$

where $I(r)$ is a binary function whose value is 1 when the file at rank r is a relevant file, and 0 otherwise. The parameter RT in the summation bound for the total number of highest-ranked files that are examined for the calculation of AP for a given query Q. The denominator rel_Q is the total number of relevant files in the collection for Q. AP estimates the area under the precision-recall curve and therefore it is suitable for comparing the ranking algorithms [21]. MAP is computed by taking the mean of the average precisions for all the queries. In addition to these metrics; we also present the number of hits ($H@r$) for the bug reports [23], which gives the number of bugs for which at least one relevant source file is retrieved in the ranked lists above a certain cut-off point r.

We used MAP for comparing the different retrieval methods as it is the most comprehensive metric that takes into account both the precision and the recall at multiple ranks. In computation of this metric for the results we report in this paper, we set $RT = 100$ in the summation in Eq. (9). In order to evaluate whether the improvements obtained with the proposed approaches are significant or not, we used pairwise student's t-test on the average precisions for the queries.

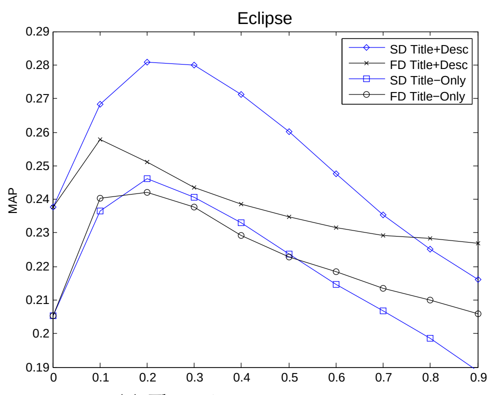
4.3 Bug Localization Experiments

For an in-depth analysis of the retrievals, we divide each bug report into two parts, namely *Title* and *Description*. We first conduct two sets of experiments using these two parts for each bug report *without* Query Conditioning (QC): (1) Retrievals with MRF modeling using only the titles of the bug reports. The queries used for these retrieval are denoted "title-only". And (2) Retrieval with MRF modeling using the complete bug reports, that is, including both the titles and the descriptions for the bug reports. The queries used for these retrievals are denoted "title+desc". Then, we incorporate QC in the second category of retrievals and analyze the usefulness of including stack traces and patches in queries by comparing the overall retrieval accuracy for the set of bug reports that contain these elements to the remaining set of the bug reports in our query sets.

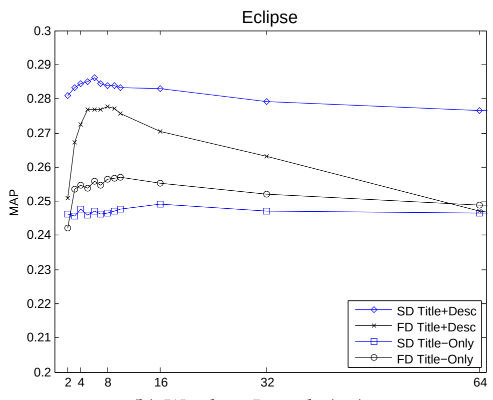
4.3.1 Parameter Sensitivity Analysis

The model parameters that affect the quality of the retrievals in our retrieval framework are: (1) The window length parameter (W). (2) The mixture parameters of the respective dependency models (λ_{SD}, λ_{FD}). And (3) The Dirichlet smoothing parameter (μ). While W sets the upper bound for the number of intervening terms between the terms of the 3-node cliques, λ_{SD} and λ_{FD} simply adjust the amount of interpolation of the scores obtained with the 2-node cliques with those obtained with the 3-node cliques as explained in Section 3. As the ranking is invariant to a constant scaling in the mixture parameters, we enforce the constraints $\lambda_{FI} + \lambda_{SD} = 1$ and $\lambda_{FI} + \lambda_{FD} = 1$ for the SD and the FD modeling, respectively. In all experiments, we empirically set the Dirichlet smoothing parameter as $\mu = 4000$. Note that the retrieval accuracy is not very sensitive to the variations on this parameter [7].

Fig. 6 and 7 plot the retrieval accuracies for bug localization in terms of MAP as the window length and the mixture parameters are varied for the "title-only" and the "title+desc" queries. As shown in Fig. 6a and 7a, a value of 0.2 consistently works well for the mixture parameters in general. Note that when $\lambda_{SD} = \lambda_{FD} = 0.0$, SD and FD use only the 2-node cliques hence they reduce to FI, the Smoothed Unigram Model (SUM). As for the window length, Fig. 6b and 7b illustrate the effect of varying this parameter for $\lambda_{SD} = \lambda_{FD} = 0.2$. On

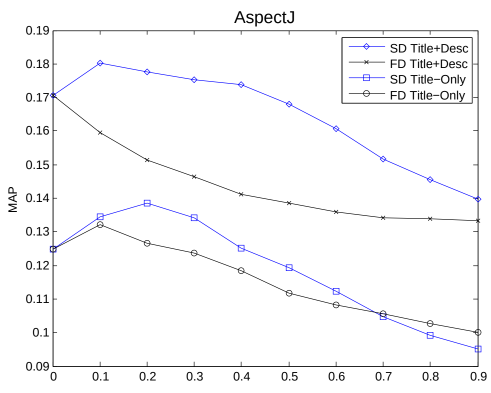


(a) The mixture parameters

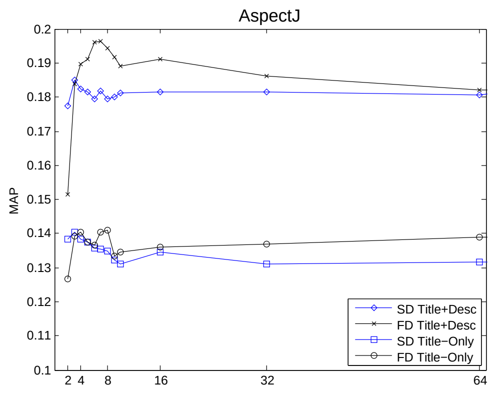


(b) Window Length (W)

Fig. 6. *The effects of varying model parameters on MAP for Eclipse. The figure on the left shows the MAP values as the mixture parameters (λ_{SD} for SD assumption and λ_{FD} for FD assumption) are varied while the window length parameter is fixed as $W = 2$. The figure on the right shows the MAP values as W is varied while the mixture parameters are fixed at 0.2.*



(a) The mixture parameters



(b) Window Length (W)

Fig. 7. *The effects of varying model parameters on MAP for AspectJ. Same as Fig. 6*

average, $W = 8$ results in the best retrieval accuracies for the analyzed projects for both types of queries.

As shown in Fig. 6a and 7a, when the window length is set as $W = 2$, SD performs better than FD in all experiments across the projects. This is because the terms are required to be adjacent to be matched in the code base when we use this setting and therefore the order of the terms becomes more important. Interestingly, as the window length increases, FD catches up with SD. Overall, SD is less sensitive to the window length parameter, achieving high retrieval accuracies consistently.

4.3.2 Retrieval Results

In this section, we compare the retrieval performances of the dependence models (SD and FD) to the Full Independence (FI) model. We fix the interpolation parameters as $\lambda_{SD} = \lambda_{FD} = 0.2$ and the window lengths as $W = 8$ in all the experiments presented in the remainder of this paper.

Table 4 presents the bug localization accuracies on the evaluated projects for the "title-only" queries and MRF modeling. With these experiments we explore the retrieval accuracy of Terrier+ for short queries comprising only a few terms. The last row of the table shows the "baseline" accuracy; this is obtained with the FI assumption, which, as mentioned previously, is the same thing as the Smoothed Unigram Model (SUM) [10], [7], [9]. The highest score in each column is shown in bold. All the improvements reported in this table obtained with the dependency models over FI are statistically significant at $\alpha = 0.05$ level. Note that incorporating the term dependencies into the retrievals improves the accuracy of bug localization substantially in terms of the 6 metrics presented in the table.

Table 5 presents the bug localization accuracies for the "title+desc" queries without QC. That is, the entire textual content of the bug reports, without any query conditioning, is used in querying the code base. The reported improvements obtained with FD and SD over FI are also statistically significant at $\alpha = 0.05$ in this table. Note that the retrieval accuracies improve significantly when the description parts of the bug reports are also included in retrievals (even though we did not include QC). While SD and FD perform comparably well in these experiment on the Eclipse project, FD outperforms SD on AspectJ on average in terms of MAP.

We are now in a position to answer the first of the five research questions (RQ) formulated in Section 2 of this

TABLE 4
Retrieval accuracy with the "title-only" queries. (We treat FI as baseline since it is synonymous with SUM.)

Method	Eclipse						AspectJ					
	MAP	P@1	P@5	R@5	R@10	H@10	MAP	P@1	P@5	R@5	R@10	H@10
FD	**0.2564 (+24.83%)**	**0.2198**	**0.1110**	**0.3199**	**0.4070**	**2,100**	**0.1410 (+12.89%)**	**0.1409**	**0.0832**	**0.1794**	**0.2420**	124
SD	0.2466 (+20.06%)	0.2116	0.1069	0.3083	0.3934	2,042	0.1348 (+7.93%)	0.1340	0.0790	0.1675	0.2382	**125**
FI	0.2054	0.1710	0.0883	0.2556	0.3417	1,805	0.1249	0.1375	0.0708	0.1498	0.2079	111

TABLE 5
Retrieval accuracy for the "title+desc" queries. (We treat FI as baseline since it is synonymous with SUM.)

Method	Eclipse						AspectJ					
	MAP	P@1	P@5	R@5	R@10	H@10	MAP	P@1	P@5	R@5	R@10	H@10
FD	0.2778 (+16.87%)	0.2496	0.1201	0.3427	0.4317	2,249	**0.1945 (+14.08%)**	**0.2131**	**0.0997**	**0.2322**	0.2996	142
SD	**0.2840 (+19.48%)**	**0.2543**	**0.1232**	**0.3530**	**0.4391**	**2,268**	0.1794 (+5.22%)	0.1856	0.0990	0.2203	**0.3096**	**148**
FI	0.2377	0.2020	0.1060	0.3046	0.3859	2,044	0.1705	0.1856	0.0880	0.2003	0.2693	126

TABLE 6
Retrieval accuracy for the "title+desc" queries with Query Conditioning (QC). (We treat FI as baseline since it is synonymous with SUM.)

Method	Eclipse						AspectJ					
	MAP	P@1	P@5	R@5	R@10	H@10	MAP	P@1	P@5	R@5	R@10	H@10
FD	**0.3019 (+17.93%)**	0.2696	0.1274	0.3709	**0.4640**	**2,386**	**0.2307 (+8.31%)**	**0.2715**	**0.1155**	**0.2703**	0.3400	161
SD	0.3014 (+17.74%)	**0.2704**	**0.1290**	**0.3749**	0.4599	2,354	0.2263 (+6.24%)	0.2646	0.1148	0.2658	**0.3554**	**167**
FI	0.2560	0.2186	0.1114	0.3263	0.4102	2,147	0.2130	0.2440	0.1052	0.2438	0.3215	147

paper. For convenience, here is the question again:

RQ1: *Does including code proximity and order improve the retrieval accuracy for bug localization. If so, to what extend?*

Based on the results presented in Tables 4, 5 and 6, we conclude that incorporating the spatial code proximity and order into the retrievals improves the accuracy of automatic bug localization significantly. On average, for both short and long queries which may contain stack traces and/or patches, SD and FD modeling consistently enhance the retrieval performance of Terrier+ over FI across the projects. The improvements are up to 24.83% for the Eclipse project and up to 14.08% for the AspectJ project in terms of MAP.

4.3.3 The effect of QC on Retrievals

The retrieval accuracies obtained with QC on the "title+desc" queries are presented in Table 6 where each bug report is first probed for stack traces and patches in order to extract the most useful source code identifiers to be used in bug localization as explained in Section 3.2. The results shown in Table 6 help us answer the following research question that was presented in Section 2:

RQ2: *Is QC effective on improving the query representation vis-a-vis the source code?*

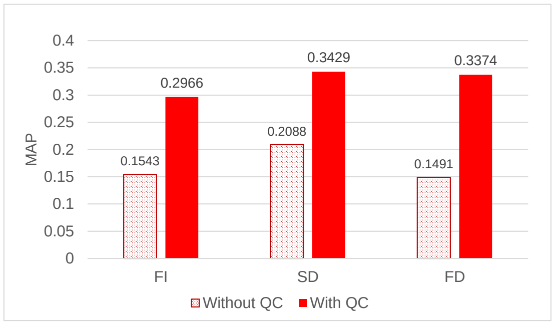
Comparing the results presented in Tables 5 and 6, we conclude that QC indeed leads to superior query formulation for source code retrieval. For all three forms of the dependency assumption, we obtained significant improvements with QC in terms of the 6 evaluation metrics mentioned in the tables.

The main benefits of QC are seen for the bug reports that contain stack traces since patches are included only in a few bug reports. Fig. 8 presents the retrieval accuracies of Terrier+ obtained specifically with the bug reports that contain stack traces. The figure shows that accuracy of the retrievals doubles on average with QC for both projects in term of MAP, reaching values above the 0.3 threshold.

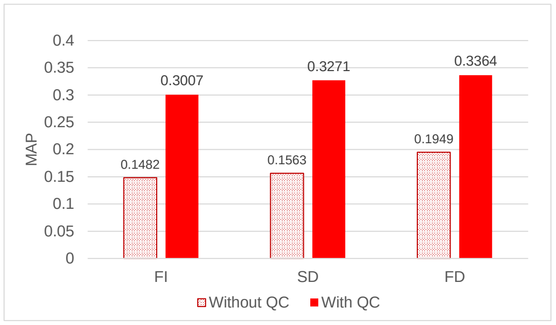
Fig. 8 also demonstrates the effect of the MRF modeling with stack traces. Comparing the results obtained with the dependency models, we observe that FD and SD outperform FI consistently when QC is used in retrievals with the stack traces. The main reason for these results is that the order and the proximity of the terms in stack traces are extremely important in locating the relevant source files. As we also mentioned in Section 3, the likelihood of a file to be relevant to a query increases when it contains longer phrases from the stack trace with the same order and proximity relationships. Interestingly, when the queries are not processed with QC, the average retrieval accuracy with FD on the Eclipse project is slightly lower than FI. This is clearly due to the noise in the lengthy stack traces that contain many method signatures most of which are irrelevant to the bug. The QC framework effectively removes the irrelevant method signatures from the trace for a better query representation.

4.3.4 The Role of Stack Traces in Automatic Bug Localization

Studies haven shown that the developers who are in charge of fixing bugs look for stack traces, test cases

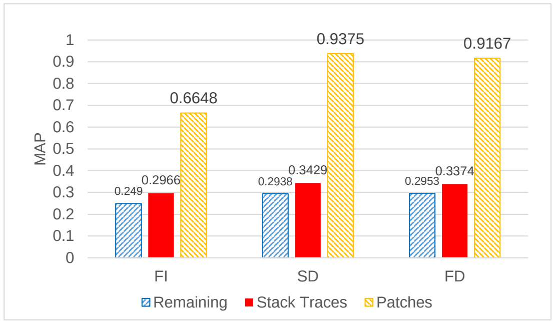


(a) Eclipse

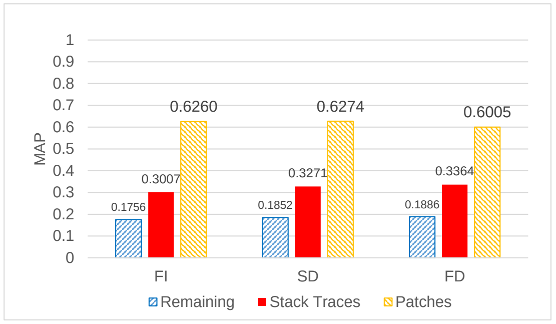


(b) AspectJ

Fig. 8. *The Effect of Query Conditioning (QC) on bug localization with bug reports containing stack traces.*



(a) Eclipse



(b) AspectJ

Fig. 9. *The effect of including structural elements in bug reports on automatic bug localization accuracy. Patches lead to the highest retrieval scores while the bug reports with no stack traces or patches perform the worst in terms of MAP.*

and the steps contained therein, in order to reproduce the bugs, these being the most useful structural elements for comprehending the underlying cause of the bugs and fixing them [14], [15]. Among these structural elements, stack traces are very important for the work we report in this paper. They are not only frequently included in bug reports but also a good source of discriminative source code identifiers for automatic bug localization. Fig. 9 presents the retrieval accuracies obtained with the bug reports containing different types of structural elements. In the figure, "remaining" denotes the bug reports that do not contain any stack traces or patches.

That sets us up to answer the research question RQ3 that was previously articulated in Section 2:

RQ3: *Does including stack traces in the bug reports improve the accuracy of automatic bug localization?*

As demonstrated in Fig. 9, bug reports with patches lead to the highest accuracies as expected. After the patches, stack traces hold the second position in terms of their usefulness in locating the relevant source code. The retrieval results for the remaining bug reports that do not contain any stack traces or patches are the worst. Based on these results, we conclude that including stack traces in the bug reports does improve the bug localization accuracy. Note that the retrieval accuracies we obtained with the stack traces are consistently above the 0.3 threshold for the analyzed projects in terms of MAP.

4.4 Comparison to Automatic Query Reformulation (QR)

In [13], we proposed an automatic Query Reformulation (QR) framework to improve the query representation for bug localization. For experimental evaluation, we used the title of a bug report as an initial query which is reformulated via Pseudo Relevance Feedback based on the retrieval results obtained with the initial query. The experimental evaluation of the approach showed that the proposed Spatial Code Proximity (SCP) based QR model outperforms the state-of-the-art QR models. That admittedly very brief introduction to SCP leads us to the next question that was posed originally in Section 2:

RQ4: *How does the MRF-based retrieval framework compare with the QR-based retrieval framework for bug localization?*

Comparing the retrieval accuracies presented in Table 7 and Table 8 to the retrieval accuracies of the QR models reported in [13], we observe that MRF framework outperforms the SCP-based QR (denoted as SCP-QR in the tables) on the average. For the Chrome project, while the differences between the average precisions obtained with the respective models are not statistically significant at $\alpha = 0.05$, the differences in terms of the presented recall metrics are. Additionally, H@10 values obtained with the MRF framework are considerably higher than the values obtained with the SCP-QR. For the Eclipse project, both SD and FD performs better than SCP-QR in terms of the reported metrics. The differences are statistically

TABLE 7
QR vs. MRF on Eclipse with the "title-only" queries.

Method	MAP	P@1	P@5	R@5	R@10	H@10
FD	**0.2564**	**0.2198**	**0.1110**	**0.3199**	**0.4070**	**2,100**
SD	0.2466	0.2116	0.1069	0.3083	0.3934	2,042
SCP-QR	0.2296	0.1906	0.1014	0.2853	0.3746	1,915

TABLE 8
QR vs. MRF on Chrome with the "title-only" queries.

Method	MAP	P@1	P@5	R@5	R@10	H@10
FD	**0.1951**	**0.1844**	**0.1061**	**0.2394**	**0.3159**	**178**
SD	0.1814	0.1760	0.1039	0.2288	0.3137	177
SCP-QR	0.1820	0.1788	0.0933	0.2021	0.2775	151

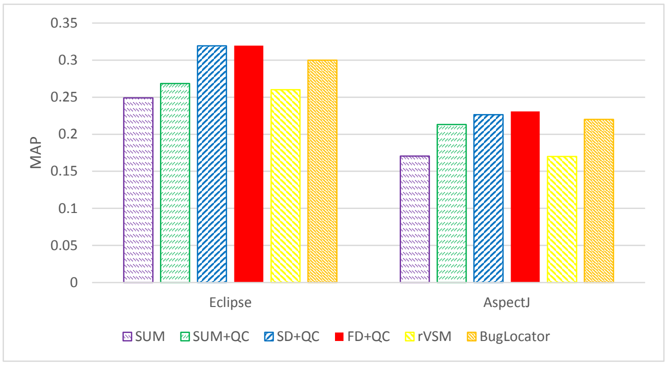


Fig. 10. *Comparison of the retrieval models for Bug Localization*

significant at $\alpha = 0.05$.

4.5 Comparison with Bug Localization Techniques That Use Prior Development History

Another important class of IR approaches to bug localization is based on the prior development history [7], [9]. In [9], Zhou et al. proposed BugLocator, a retrieval tool that uses the textual similarities between a given bug report and the prior bug reports to enhance the bug localization accuracy. The main motivation behind BugLocator is that the same files tend to get fixed for similar bug reports during the life-cycle of a software project. Another study that leverages the past development efforts was reported by us in [7]. In that work, we mined the software repositories for the defect and modification likelihoods of the source files in order to estimate a prior probability distribution which could then be used for a more accurate source code retrieval for bug localization. This brief review of this class of approaches to bug localization takes us to the last of the research questions stated in Section 2:

RQ5: *How does the MRF-based retrieval framework compare with the other bug localization frameworks that leverage the past development history?*

The accuracy of BugLocator is also evaluated on Eclipse v3.1 and iBugs datasets. The evaluations on the Eclipse project are performed using 3,075 bug reports filed for the version 3.1 while our experiments with Terrier+ were performed using 4,035 bug reports filed for the same version. In order to compare the performance of Terrier+ to that of BugLocator, we repeated the experiments using only the bug reports with which the BugLocator was evaluated.

Using the MAP metric, Fig. 10 shows the accuracy of the proposed framework along with that of BugLocator. In the figure, we also included the accuracies obtained with the revised Vector Space Model (rVSM) [9] that, according to the authors of BugLocator, yields retrieval results superior to those obtained with the classic Vector Space Model (VSM). As can be seen in the figure, FD+QC and SD+QC outperform BugLocator with MAP values above 0.32 threshold for the Eclipse project. In comparison, BugLocator performs better than SUM and SUM+QC with a MAP value of 0.30, while SUM+QC outperforms rSVM. The performance comparisons with the different models are similar for the AspectJ project.

The other bug localization technique that reports improved retrieval performance with past development history leverages the version histories [7]. In this study, we showed that TFIDF model incorporating defect histories of the software artifacts (TFIDF+DHbPd) reaches a MAP value of 0.2258 on the AspectJ project.

Based on these results, we conclude that *Terrier+ performs better than these other bug localization techniques without having to leverage the past development history for a software library.*

5 Relevant Work

Traditional methods for bug localization rely on the dynamic or the static properties of software [24], [25], [26], [27], [28], [29]. Whereas dynamic approaches require that a set of test cases be executed to locate the parts of the program causing the bug, static approaches aim to leverage the static properties of the software such as its function call graphs, dependency relationships between the code segments, etc. The main problem with the static approaches is that they tend to return too many false positives [30]. Although dynamic approaches tend to be more accurate than static methods, designing an exhaustive set of test cases that could effectively be used to reveal defective behaviors is very difficult and expensive. The bug localization approach we presented in this paper does not require the execution of a program, not to speak of the fact that it is also lightweight and inexpensive.

Concept location, feature/concern location and bug localization are closely related problems in software engineering. Early work on using text retrieval methods for concept location includes the work by Marcus et al. [1]. They used Latent Semantic Indexing (LSI) to retrieve the software artifacts in response to short queries. The retrievals are performed in the lower dimensional LSI space which assigns greater importance to the terms that frequently co-occur in the source files. This framework can also be used to expand a given initial query that consists of a single query term initially. In [31], Poshyvanyk et al. extended this approach to include formal concept analysis. They showed that the irrelevant search results returned by the LSI model can be reduced with formal concept analysis. Hybrid methods that combine dynamic analysis with Information Retrieval (IR) have also been proposed in this area [3], [32].

In [33], Hill et al. leveraged source code identifiers to automatically extract the phrases relevant to a given initial query. These phrases were then used to either find the relevant program elements or to manually reformulate the query for superior feature/concern localization. In [34], Hill et al. investigated the effect of the position of a query term on the accuracy of the search results. The main idea behind this study is that the location of a query term in the method signatures and in the method bodies determines its importance in the search process.

In [35], Gay et al. used Explicit Relevance Feedback for Query Reformulation (QR) for the purpose of concept location. This framework requires developers to engage in an iterative query/answer session with the search engine. At each iteration, the developer is expected to judge the relevance of the returned results vis-á-vis the current query. Based on these judgments, the query is reformulated with the Rocchio's formula [36] and resubmitted to obtain the next round of retrieval results. This process is repeated until the target file is located or the developer gives up.

In [37], Haiduc et al. introduced *Refocus*, an automatic QR tool for text retrieval in software engineering. Refocus automatically reformulates a given query by choosing the best QR technique which is determined by training a decision tree on a separate query set and their retrieval results. After training, based on the statistics of the given query, the decision tree recommends an automatic query reformulation technique that is expected to perform the best among the others.

Recently, several studies have investigated the Information Retrieval (IR) algorithms for the retrieval of software artifacts for bug localization. In a comparative study, Rao and Kak evaluated a number of generic and composite IR models to localize the files that should be fixed to resolve bugs [10]. The main result that came out of this study was that simpler models, such as the VSM or the Smoothed Unigram Model, performed better than the more sophisticated models such as LDA (Latent Dirichlet Allocation). Another similar comparative study is by Lukins et al. [2]. Their results show LDA performing at least as well as LSA. In a related contribution, Nguyen et al. [16] also proposed an LDA-based approach to narrow down the search space for improving bug localization accuracy. In [8], Ashok et al. have shown how the relationship graphs can be used to retrieve source files and prior bugs in response to what they refer to as "fat queries" that consist of structured and unstructured data.

6 Threats to Validity

The threats to the validity of our approach mainly emanate from the scope of the datasets we used in our evaluations and the procedures used to create them. Although our experimental evaluation involves large open source projects commonly used in the evaluation of bug localization approaches, the performance of the proposed framework may vary for other open source or propriety projects. An important step used in the preparation of these datasets is the reconstruction of the links between the bug tracking databases and the corresponding development effort in the software repositories. This reconstruction step is performed using regular expressions to link the bug reports to the repository commits based on the commit messages. Despite the fact that a large number of bug reports are accurately linked to the repository commits, this linking process may occasionally fail when the commit messages in the versioning tools are much too cryptic for regular-expression based matching to succeed [38].

7 Conclusions

We presented a theoretically sound IR framework for automatic bug localization that takes into account the spatial code proximity and term ordering relationships in a code base for improved retrieval accuracy. The proposed retrieval framework benefits from a fuzzy matching mechanism between the term blocks of the queries and the source code. At the heart of the our approach is the concept of Markov Random Fields that captures both the positional and the order attributes of the terms in source files. Our experimental validation involving large open-source software projects and over 4,000 bugs has established that the retrieval performance improves significantly when both the proximity and ordering relationships between the terms are taken into account.

Our experimental evaluation also demonstrates that in conjunction with the MRF model, the proposed Query Conditioning (QC) approach effectively exploits the different types of structural information that is frequently included in bug reports. We showed that the structural elements in bug reports — particularly stack traces — contain vital information that is not well represented via-a-vis the source code with the widely used bag-of-word assumption.

Overall, on the basis of retrieval accuracies, it is clear that one can obtain significantly higher bug localization accuracies when MRF modeling and query conditioning is included in a retrieval framework compared to all other state-of-the-art approaches, even including those that take prior development history into account.

Acknowledgments

This work was supported by Infosys.

References

[1] A. Marcus, A. Sergeyev, V. Rajlich, and J. Maletic, "An information retrieval approach to concept location in source code," in *Reverse Engineering, 2004. Proceedings. 11th Working Conference on*, nov. 2004, pp. 214 – 223.

[2] S. Lukins, N. Kraft, and L. Etzkorn, "Source code retrieval for bug localization using latent dirichlet allocation," in *Reverse Engineering, 2008. WCRE'08. 15th Working Conference on*. IEEE, 2008, pp. 155–164.

[3] D. Liu, A. Marcus, D. Poshyvanyk, and V. Rajlich, "Feature location via information retrieval based filtering of a single scenario execution trace," in *Proceedings of the twenty-second IEEE/ACM international conference on Automated software engineering*. ACM, 2007, pp. 234–243.

[4] A. D. Lucia, F. Fasano, R. Oliveto, and G. Tortora, "Recovering traceability links in software artifact management systems using information retrieval methods," *ACM Transactions on Software Engineering and Methodology (TOSEM)*, vol. 16, no. 4, p. 13, 2007.

[5] D. Poshyvanyk, A. Marcus, R. Ferenc, and T. Gyimóthy, "Using information retrieval based coupling measures for impact analysis," *Empirical Software Engineering*, vol. 14, no. 1, pp. 5–32, 2009.

[6] M. Gethers, B. Dit, H. Kagdi, and D. Poshyvanyk, "Integrated impact analysis for managing software changes," in *Software Engineering (ICSE), 2012 34th International Conference on*. IEEE, 2012, pp. 430–440.

[7] B. Sisman and A. Kak, "Incorporating version histories in information retrieval based bug localization," in *Mining Software Repositories (MSR), 2012 9th IEEE Working Conference on*. IEEE, 2012, pp. 50–59.

[8] B. Ashok, J. Joy, H. Liang, S. K. Rajamani, G. Srinivasa, and V. Vangala, "Debugadvisor: a recommender system for debugging," in *Proceedings of the the 7th joint meeting of the European software engineering conference and the ACM SIGSOFT symposium on The foundations of software engineering*. ACM, 2009, pp. 373–382.

[9] 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 *Software Engineering (ICSE), 2012 34th International Conference on*. IEEE, 2012, pp. 14–24.

[10] S. Rao and A. Kak, "Retrieval from software libraries for bug localization: a comparative study of generic and composite text models," in *Proceeding of the 8th working conference on Mining software repositories*, 2011, pp. 43–52.

[11] N. Bettenburg, R. Premraj, T. Zimmermann, and S. Kim, "Extracting structural information from bug reports," in *Proceedings of the 2008 international working conference on Mining software repositories*. ACM, 2008, pp. 27–30.

[12] D. Metzler and W. Croft, "A markov random field model for term dependencies," in *Proceedings of the 28th annual international ACM SIGIR conference on Research and development in information retrieval*. ACM, 2005, pp. 472–479.

[13] B. Sisman and A. C. Kak, "Assisting code search with automatic query reformulation for bug localization," in *Proceedings of the 10th Working Conference on Mining Software Repositories*, ser. MSR '13. Piscataway, NJ, USA: IEEE Press, 2013, pp. 309–318. [Online]. Available: http://dl.acm.org/citation.cfm?id=2487085.2487145

[14] N. Bettenburg, S. Just, A. Schröter, C. Weiss, R. Premraj, and T. Zimmermann, "What makes a good bug report?" in *Proceedings of the 16th ACM SIGSOFT International Symposium on Foundations of software engineering*. ACM, 2008, pp. 308–318.

[15] A. Schroter, N. Bettenburg, and R. Premraj, "Do stack traces help developers fix bugs?" in *Mining Software Repositories (MSR), 2010 7th IEEE Working Conference on*. IEEE, 2010, pp. 118–121.

[16] 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 26th IEEE/ACM International Conference on Automated Software Engineering (ASE'11)*, nov. 2011, pp. 263 –272.

[17] D. Kollar and N. Friedman, *Probabilistic graphical models: principles and techniques*. The MIT Press, 2009.

[18] J. Peng, C. Macdonald, B. He, V. Plachouras, and I. Ounis, "Incorporating term dependency in the dfr framework," in *Proceedings of the 30th annual international ACM SIGIR conference on Research and development in information retrieval*. ACM, 2007, pp. 843–844.

[19] R. Wu, H. Zhang, S. Kim, and S. Cheung, "Relink: recovering links between bugs and changes," in *SIGSOFT FSE*, 2011, pp. 15–25.

[20] V. Dallmeier and T. Zimmermann, "Extraction of bug localization benchmarks from history," in *Proceedings of the twenty-second IEEE/ACM international conference on Automated software engineering*. ACM, 2007, pp. 433–436.

[21] C. Manning, P. Raghavan, and H. Schutze, *Introduction to information retrieval*. Cambridge University Press Cambridge, 2008, vol. 1.

[22] C. Macdonald, B. He, V. Plachouras, and I. Ounis, "University of glasgow at trec 2005: Experiments in terabyte and enterprise tracks with terrier," in *Proceedings of TREC 2005*, 2005.

[23] A. T. Nguyen, T. T. Nguyen, J. Al-Kofahi, H. V. Nguyen, and T. N. Nguyen, "A topic-based approach for narrowing the search space of buggy files from a bug report," in *26th International Conference on Automated Software Engineering (ASE'11)*. IEEE, 2011, pp. 263–272.

[24] J. A. Jones, M. J. Harrold, and J. Stasko, "Visualization of test information to assist fault localization," in *Proceedings of the 24th International Conference on Software Engineering*, ser. ICSE '02. New York, NY, USA: ACM, 2002, pp. 467–477. [Online]. Available: http://doi.acm.org/10.1145/581339.581397

[25] V. Dallmeier, C. Lindig, and A. Zeller, "Lightweight bug localization with ample," in *Proceedings of the 6th international symposium on Automated analysis-driven debugging*. ACM, 2005, pp. 99–104.

[26] B. Liblit, M. Naik, A. X. Zheng, A. Aiken, and M. I. Jordan, "Scalable statistical bug isolation," *ACM SIGPLAN Notices*, vol. 40, no. 6, pp. 15–26, 2005.

[27] C. Liu, L. Fei, X. Yan, J. Han, and S. P. Midkiff, "Statistical debugging: A hypothesis testing-based approach," *Software Engineering, IEEE Transactions on*, vol. 32, no. 10, pp. 831–848, 2006.

[28] D. Hovemeyer and W. Pugh, "Finding bugs is easy," *ACM SIGPLAN Notices*, vol. 39, no. 12, pp. 92–106, 2004.

[29] M. P. Robillard, "Topology analysis of software dependencies," *ACM Transactions on Software Engineering and Methodology (TOSEM)*, vol. 17, no. 4, p. 18, 2008.

[30] B. Dit, M. Revelle, M. Gethers, and D. Poshyvanyk, "Feature location in source code: a taxonomy and survey," *Journal of Software: Evolution and Process*, vol. 25, no. 1, pp. 53–95, 2013.

[31] D. Poshyvanyk, M. Gethers, and A. Marcus, "Concept location using formal concept analysis and information retrieval," *ACM Transactions on Software Engineering and Methodology (TOSEM)*, vol. 21, no. 4, p. 23, 2012.

[32] D. Poshyvanyk, Y.-G. Guéhéneuc, A. Marcus, G. Antoniol, and V. Rajlich, "Combining probabilistic ranking and latent semantic indexing for feature identification," in *14th IEEE International Conference on Program Comprehension, 2006*. IEEE, 2006, pp. 137–148.

[33] E. Hill, L. Pollock, and K. Vijay-Shanker, "Automatically capturing source code context of nl-queries for software maintenance and reuse," in *31st International Conference on Software Engineering, 2009. ICSE 2009.*, 2009, pp. 232–242.

[34] ——, "Improving source code search with natural language phrasal representations of method signatures," in *Proceedings of the 2011 26th IEEE/ACM International Conference on Automated Software Engineering*. IEEE Computer Society, 2011, pp. 524–527.

[35] G. Gay, S. Haiduc, A. Marcus, and T. Menzies, "On the use of relevance feedback in IR-based concept location," in *Software Maintenance, 2009. ICSM 2009. IEEE International Conference on*. IEEE, 2009, pp. 351–360.

[36] J. Rocchio, "Relevance feedback in information retrieval," 1971.

[37] S. Haiduc, G. Bavota, A. Marcus, R. Oliveto, A. De Lucia, and T. Menzies, "Automatic query reformulations for text retrieval in software engineering," in *Proceedings of the 2013 International Conference on Software Engineering*, ser. ICSE '13. Piscataway, NJ, USA: IEEE Press, 2013, pp. 842–851. [Online]. Available: http://dl.acm.org/citation.cfm?id=2486788.2486898

[38] A. Bachmann, C. Bird, F. Rahman, P. Devanbu, and A. Bernstein, "The missing links: Bugs and bug-fix commits," in *Proceedings of the eighteenth ACM SIGSOFT international symposium on Foundations of software engineering*. ACM, 2010, pp. 97–106.

Bunyamin Sisman Bunyamin Sisman received his BS degree in Computer Engineering from Istanbul Technical University, Istanbul, Turkey in 2006. He received his MS degree in Computer Engineering from Purdue University in 2009. Since then he has been pursuing his PhD studies in Computer Engineering at Purdue University. His research interests include Source Code Retrieval from Large Software Repositories and Mining Software Repositories for Bug Localization.

Avinash C. Kak is a professor of electrical and computer engineering at Purdue University and a consultant to Infosys. His book Programming with Objects (John Wiley & Sons, 2003) is used by a number of leading universities as a text on object-oriented programming. His most recent book Scripting with Objects (John Wiley & Sons, 2008) focuses on object-oriented scripting. These are two of the three books for an Objects Trilogy he is creating. The last, expected to be finished sometime later this year, is titled Designing with Objects. In addition to computer languages and software engineering, his research interests include sensor networks, computer vision, and robotic intelligence. His coauthored book Principles of Computerized Tomographic Imaging was republished as a Classic in Applied Mathematics by SIAM.