2014 IEEE International Conference on Software Maintenance and Evolution

Compositional Vector Space Models for Improved Bug Localization

Shaowei Wang[1], David Lo[1], and Julia Lawall[2]
[1]School of Information Systems, Singapore Management University
[2]Inria/LIP6-Whisper/UPMC/Sorbonne University
{shaoweiwang.2010,davidlo}@smu.edu.sg, julia.lawall@lip6.fr

***Abstract*—Software developers and maintainers often need to locate code units responsible for a particular bug. A number of Information Retrieval (IR) techniques have been proposed to map natural language bug descriptions to the associated code units. The vector space model (VSM) with the standard tf-idf weighting scheme ($VSM_{natural}$), has been shown to outperform nine other state-of-the-art IR techniques. However, there are multiple VSM variants with different weighting schemes, and their relative performance differs for different software systems.**

Based on this observation, we propose to compose various VSM variants, modelling their composition as an optimization problem. We propose a genetic algorithm (GA) based approach to explore the space of possible compositions and output a heuristically near-optimal composite model. We have evaluated our approach against several baselines on thousands of bug reports from AspectJ, Eclipse, and SWT. On average, our approach ($VSM_{composite}$) improves hit at 5 (Hit@5), mean average precision (MAP), and mean reciprocal rank (MRR) scores of $VSM_{natural}$ by 18.4%, 20.6%, and 10.5% respectively. We also integrate our compositional model with $AmaLgam$, which is a state-of-art bug localization technique. The resultant model named $AmaLgam_{composite}$ on average can improve Hit@5, MAP, and MRR scores of $AmaLgam$ by 8.0%, 14.4% and 6.5% respectively.

I. Introduction

Maintenance tasks can account for up to 90% of the overall cost of a software project [1]. It is furthermore estimated that up to 50% of maintenance cost is incurred to comprehend the existing code [2], [3], especially for large systems that are maintained by many developers. To reduce the cost of program comprehension and software maintenance, various techniques have been proposed [4]–[11]. One family of techniques is called bug localization that maps a bug report to relevant buggy files [5], [12]–[14].

As bug reports are often expressed in natural language, many Information Retrieval (IR) techniques have been used to perform bug localization. The vector space model (VSM) with the standard tf-idf weighting scheme has been shown to outperform many other non-VSM information retrieval techniques (e.g., LDA, LSA, etc.) [15]. However, there are many well-known variants of VSM with different weighting schemes [16]. In this paper, we leverage this fact by composing VSMs with different tf-idf weighting schemes into a more effective composite model. The weaknesses of VSM with the standard tf-idf thus could be addressed or shadowed by the strengths of other VSMs with different tf-idf weighting schemes. This is similar to the boosting and ensemble approach in data mining [17].

There are various ways to compose multiple VSMs with various tf-idf weighting schemes. Some compositions may improve accuracy as compared to the individual models; while other compositions could perform worse. The various ways to compose the VSMs with different tf-idf weighting schemes form a search space, and we would like to find the composition that gives the highest accuracy within this space. Unfortunately, this search space is very large, as there are many ways to compose the various VSMs.

The problem of finding the best composition amounts to an optimization problem: Given a training dataset, search for a near-optimal composition that performs best on the training data. This near-optimal composition can then be used to localize new bug reports (i.e., find the buggy files). Researchers in the machine learning and meta-heuristics communities have proposed various search-based optimization algorithms, *e.g.*, simulated annealing [18] and genetic algorithms [19], that could be used to find a heuristically near-optimal solution.

In this work, we propose a search-based compositional bug localization engine that works in two phases: training and deployment. In the training phase, the engine takes in a training set of bug reports, which are textual documents, along with the files that are modified to fix the bugs. This training set is then used to evaluate a particular composition. A good composition should be able to locate the buggy files of many bug reports in the training set with a high accuracy. To identify promising compositions, search heuristics are employed to traverse the search space to find near-optimal compositions that perform best on the training set. We compose 15 VSM variants, each with a different tf-idf weighting scheme, to form a new composite model. In the deployment phase, the composite model is used to localize new bugs.

We test our search-based bug localization engine, named $VSM_{composite}$ on thousands of bug reports from three software projects, namely AspectJ, Eclipse, and SWT. We give a bug report as input and use a bug localization tool to retrieve the relevant buggy files. Our experiment results show that $VSM_{composite}$ outperforms the vector space model with the standard tf-idf weighting scheme ($VSM_{natural}$). $VSM_{composite}$ can improve the hit at 5 (Hit@5), mean average precision (MAP), and mean reciprocal rank (MRR) scores of $VSM_{natural}$ by 18.4%, 20.6%, and 10.5%, respectively. We also integrate our approach with the state-of-the-art approach $AmaLgam$ [14] that uses not only the textual descriptions in bug reports but also other pieces of information, i.e., version history, similar reports, and structure. We refer to the resulting approach as $AmaLgam_{composite}$ and demonstrate

1063-6773/14 $31.00 © 2014 IEEE
DOI 10.1109/ICSME.2014.39

IEEE computer society

that $AmaLgam_{composite}$ improves the Hit@5, MAP, and MRR scores of $AmaLgam$ by 8.0%, 14.4%, and 6.5%, respectively.

Our contributions are as follows:

1. We propose an approach to compose 15 vector space models, each with a different tf-idf weighting scheme, by leveraging a search-based algorithm. For this, we define the search space of possible compositions and adapt a genetic algorithm to heuristically find a near optimal solution.
2. We show that, on average, compared with $VSM_{natural}$, our approach ($VSM_{composite}$) could improve Hit@5, MAP, and MRR of $VSM_{natural}$ by 18.4%, 20.6%, and 10.5%, respectively.
3. We show that, on average, compared with $AmaLgam$, the enhanced approach $AmaLgam_{composite}$ improves $AmaLgam$ in terms of Hit@5, MAP, and MRR by 8.0%, 14.4%, and 6.5%, respectively.

The structure of this paper is as follows. Section II describes background information on bug localization and genetic algorithms. Section III describes the individual vector space models with various tf-idf weighting schemes, which serve as the building blocks that our engine combines to form a composite model. We elaborate on our search-based composition engine in Section IV. We introduce the details of $AlaLgam$ and how we integrate it with various VSM models in Section V. We present our evaluation in Section VI. In Section VII, we discuss related work. Finally, Section VIII concludes and mentions future work.

II. Background

In this section, we introduce the bug localization problem and search-based algorithms. We describe how the vector space model can be used for bug localization and briefly present genetic algorithms (GA).

A. Bug Localization

The goal of bug localization is to link a particular bug report to the files that are responsible for the bug. Most bug localization approaches uses information retrieval techniques to process a bug report, which is a textual document, and retrieve the names of files that are likely to contain the bug.

One IR technique that can be used for bug localization is the vector space model (VSM). VSM has been shown to outperform many other IR approaches for bug and concern localization [5], [15]. Many state-of-the-art bug localization techniques are built on top of VSM [12]–[14]. In Table I, we show an example of a bug report and the set of files that is responsible for this bug. VSM looks for files that are similar to the bug report. In this case, the files “AntEditor.java” and “AntEditorSourceViewerConfiguration.java” are similar to the bug report as they contain the words “Ant”, and “editor”. We now present the vector space model and how it can be used for bug localization.

Vector Space Model. In the vector space model (VSM), each document is represented as a vector of values. Each value in the vector represents the weight of a term in the document. One popular way to assign weights is to use the concepts of *term frequency* and *inverse document frequency* (tf-idf). The standard tf-idf scheme assigns a weight to a term t in a document d according to the formula:

TABLE I. An Eclipse Bug Report and its Buggy Source Code Files

Summary	**Ant editor** not following tab/space setting on shift right
Description	This is from 3.1 M2. I have **Ant->Editor**->Display tab width set to 2, ”Insert spaces for tab when typing” checked. I also have **Ant->Editor**->Formatter->Tab size set to 2, and ”Use tab character instead of spaces _unchecked_. Now when I open a build.xml and try to do some indentation, everything works fine according to the above settings, except when I highlight a block and press tab to indent it. It’s the tab character instead of 2 spaces that’s inserted in this case.
Fixed Files	org.eclipse.ant.internal.ui.editor.**AntEditor**.java org.eclipse.ant.internal.ui.editor.**AntEditorSourceViewerConfiguration**.java

$$weight(t, d) = tf(t, d) \times idf(t, D)$$

where t, d, D, $tf(t, d)$, $idf(t, D)$ correspond to a term, a document, a corpus (i.e., a set of documents), the frequency of t in d, and the inverse document frequency of t in D, respectively. Term frequency is the number of times a term appears in a document – it measures how important the term is to the document. Inverse document frequency captures the rarity of a term. A rare term is more useful to differentiate one document from another. To give more weight to rare terms, inverse document frequency is calculated as the logarithm of the reciprocal of the document frequency (df). The document frequency of a term t in corpus D is the number of documents in D that contain t. Besides the above definition of weight, numerous well-known variants have been proposed, depending on how term frequency and inverse document frequency are computed [16]. We present 15 variants in Section III.

Given a query and a corpus, VSM first converts the query and each document in the corpus to bags of words after doing some pre-processing steps: tokenization, identifier splitting, stop word removal, and stemming. Tokenization breaks a document into word tokens. Identifier splitting breaks a source code identifier into multiple words (e.g., the identifier “getMethodName” becomes “get”, “method”, and “name”). We use a simple CamelCase splitter [20]. Stop word removal deletes words that carry little meaning and are used in almost every document, e.g., “the”, “is”, etc. We use the list of stopwords from http://www.textfixer.com/resources/common-english-words.txt. Stemming refers to the process of reducing a word to its root form. We use the Porter stemming algorithm [21]. VSM then converts each bag of words into a vector of values by calculating the tf-idf weight of each term. Then, the cosine similarity between a query and all documents in the corpus is computed based on their representative vectors:

$$cos(q, d) = \frac{\sum\limits_{t \in (q \bigcup d)} weight(t, q) \times weight(t, d)}{\sqrt{\sum\limits_{t \in q} weight(t, q)^2} \times \sqrt{\sum\limits_{t \in d} weight(t, d)^2}}$$

where q and d are a query and a document, respectively. After computing the above scores for all documents in the corpus, we sort the documents based on their similarity scores. Finally, the top k documents are output.

In our setting, a query corresponds to a bug report or a feature enhancement request that is described in natural language, possibly including some source code snippets; a document comprises both the identifiers in the source code of a file and the comments before and within the file.

B. Search-based Algorithms

Various search-based algorithms have been proposed to solve optimization problems. Such algorithms search through the space of possible solutions. Each solution in the search space is assigned a score based on a particular objective function. The search space then forms a landscape, where some points in the landscape (i.e., solutions) are hills (i.e., they have high scores) and some others are valleys (i.e., they have low scores). An optimization technique searches for a solution with the maximal score (maximization problem), or a solution with the minimal score (minimization problem). The search space can often be very large, indeed even infinite. Various heuristics are employed to navigate through a search space to find a solution that has a good enough score or a near-optimal solution within a particular computational budget.

We now present a particular family of search-based algorithms: genetic algorithms (GA) [22]. A GA aims to maximize an objective function.

Genetic Algorithms. A genetic algorithm (GA) is a search-based algorithm that iteratively refines a set of solutions until a particular criterion is met. A solution is modeled as a chromosome, which can be represented as a bit string. A GA starts with a random population of chromosomes representing a set of solutions. It then subjects this population to several iterations of selections, crossovers, and mutations. Selection refers to the process of eliminating chromosomes with low objective scores. Crossover takes two chromosomes and swaps parts of them to produce two new chromosomes. Mutation randomly flips bits in the bit string representing a chromosome. At the end of each iteration, a new population of chromosomes (a.k.a., a new *generation*) is generated. Many iterations may be needed until a particular criterion is met - this could be a particular target objective score, a maximum number of iterations, a maximum amount of time, etc.

There are many genetic algorithm variants. Figure 1 shows the pseudocode of the one we use. Its parameters are: the number of chromosomes in a population, the number of iterations to be performed, the probability for a mutation to happen, and the probability for a crossover to happen. The parameters of such an algorithm are usually empirically determined by some preliminary experiments.

The time complexity of our GA is given by:

$$O(N_I \times (N_C \times P_C \times O(cross) + N_C \times P_M \times O(mut) + O(sel)))$$

where N_I is the number of iterations, N_C is the number of chromosomes, P_C is the crossover probability, P_M is the mutation probability, and $O(cross)$, $O(mut)$ and $O(sel)$ are the time complexities of the selection, crossover, and mutation operations respectively.

```
Procedure Genetic Algorithm
Inputs:  N_C: Number of chromosomes
         N_I: Number of iterations
         P_C: Crossover probability
         P_M: Mutation probability
Outputs: Best solution found
Method:
1: Let P = Initial population with N_C members
2: Evaluate P's members and find the best solution so far
3: Repeat N_I times
4:     P = Selection(P)
5:     P = Crossover(P, P_C)
6:     P = Mutation(P, P_M)
7:     Evaluate P and update the best solution so far
8: Output the best solution found
```

Fig. 1. Genetic Algorithm: Pseudocode

TABLE II. VARIANTS OF TF AND IDF

Term frequency	
$tf_n(t,d)$ (natural)	$\lvert\{t \lvert t \in d\}\rvert$
$tf_l(t,d)$ (logarithm)	$1 + log(tf_n)$
$tf_L(t,d)$ (Log ave)	$\frac{1+log(tf_n(t,d))}{1+log(ave_{t \in d}(tf_n(t,d)))}$
$tf_a(t,d)$ (augmented)	$0.5 + \frac{0.5 \times tf_n(t,d)}{max_t(tf_n(t,d))}$
$tf_b(t,d)$ (boolean)	$\begin{cases} 1 & if\ tf_n(t,d) > 0 \\ 0 & otherwise \end{cases}$
Document frequency	
$idf_n(t,D)$ (no)	1
$idf_l(t,D)$ (standard)	$log \frac{\lvert D\rvert}{df_t}$
$idf_r(t,D)$ (ratio)	max$\{0, log \frac{\lvert D\rvert - df_t}{df_t}\}$

The major tasks in employing a GA are to identify a mapping from the problem's potential solutions into chromosomes, and to provide an appropriate objective function.

III. VARIANTS OF THE TF-IDF WEIGHTING SCHEME

The tf-idf weight for a term in a document is the product of its term frequency score and its inverse document frequency score. There are many variants of the standard tf-idf weighting scheme, depending on how the term frequency (tf) and inverse document frequency (idf) are measured. Table II lists five variants of tf and three variants of idf. We investigate the 15 possible combinations, as shown in Table III. We discuss these tf and idf variants in more detail below.

Term frequency. Various definitions of term frequency have been proposed [16]. The simplest, $tf_n(t,d)$, which we refer to as the natural term frequency, is to use the raw frequency of a term in a document, i.e., the number of times that term t occurs in document d. However, it may be that e.g., ten occurrences of a term in a document do not truly indicate that the term is ten times more significant than another term that only appears once. Thus, other variants have been proposed. Table II shows four other variants. $tf_l(t,d)$ takes the logarithm of the natural term frequency. $tf_L(t,d)$ normalizes $tf_l(t,d)$ by dividing it by the average logarithm of the other terms in the document d. $tf_a(t,d)$ normalizes $tf_n(t,d)$ by dividing it by the frequency of the term appearing the most times in

TABLE III. VARIANTS OF THE TF-IDF WEIGHTING SCHEME. THE TF AND IDF VARIANTS ARE DESCRIBED IN TABLE II.

Name	Equation
tf_n-idf_n	$tf_n(t,d) \times idf_n(t,D)$
tf_n-idf_l	$tf_n(t,d) \times idf_l(t,D)$
tf_n-idf_r	$tf_n(t,d) \times idf_r(t,D)$
tf_l-idf_n	$tf_l(t,d) \times idf_n(t,D)$
tf_l-idf_l	$tf_l(t,d) \times idf_l(t,D)$
tf_l-idf_r	$tf_l(t,d) \times idf_r(t,D)$
tf_L-idf_n	$tf_L(t,d) \times idf_n(t,D)$
tf_L-idf_l	$tf_L(t,d) \times idf_l(t,D)$
tf_L-idf_r	$tf_L(t,d) \times idf_r(t,D)$
tf_a-idf_n	$tf_a(t,d) \times idf_n(t,D)$
tf_a-idf_l	$tf_a(t,d) \times idf_l(t,D)$
tf_a-idf_r	$tf_a(t,d) \times idf_r(t,D)$
tf_b-idf_n	$tf_b(t,d) \times idf_n(t,D)$
tf_b-idf_l	$tf_b(t,d) \times idf_l(t,D)$
tf_b-idf_r	$tf_b(t,d) \times idf_r(t,D)$

document d. $tf_b(t,d)$ ignores the actual term frequency and only considers whether or not a term appears in a document (i.e., it is a boolean model).

Inverse document frequency. Inverse document frequency (idf) is a measure of whether a term is common or rare in the documents of a corpus. A high idf value implies that a term is rare. Terms with high idf values are important to differentiate one document from another. Again, variants have been proposed [16]. We consider three variants, $idf_n(t,D)$, $idf_l(t,D)$, and $idf_r(t,D)$, defined in Table II. $idf_n(t,D)$, treats all terms as equally important, i.e., they are given the same weight. $idf_l(t,D)$, is a standard inverse document frequency, which takes the logarithm of the reciprocal of the document frequency. $idf_r(t,D)$, takes the logarithm of the ratio of documents not containing the term t and those containing the term t. The latter two variants are just two different ways to measure the rarity of a term.

Past work has shown that effective retrieval models tend to rely on a reasonable way to combine multiple retrieval signals, such as term frequency (tf) and inverse document frequency (idf) [23]. A major difficulty in developing an effective information retrieval model is that the signals (tf, idf) interact with each other in complicated ways. Thus, finding the best way to combine these signals is a long-standing challenge [23]. In this study, we do not try to understand the subtle interactions between various tf and idf variants. Instead, we simply consider the product of all the different variants of tf and idf listed in Table II. As a result, we obtain 15 variants of tf-idf weighting scheme, VSM_1, VSM_2, ..., VSM_{15}, as shown in Table III. VSM with the tf_n-idf_l weighting scheme (VSM_2) is the standard version, which is the most widely used [16]. We also refer to it as $VSM_{natural}$.

IV. SEARCH-BASED COMPOSITION ENGINE

Our search-based bug localization process is composed of two phases: training and deployment. The two phases are illustrated in Figure 2.

A. Training Phase

The training phase takes as input the vector space models described in Section III along with the training data. As training data, we take a set of bug reports, a set of files in a software project, and the set of files that are modified to fix

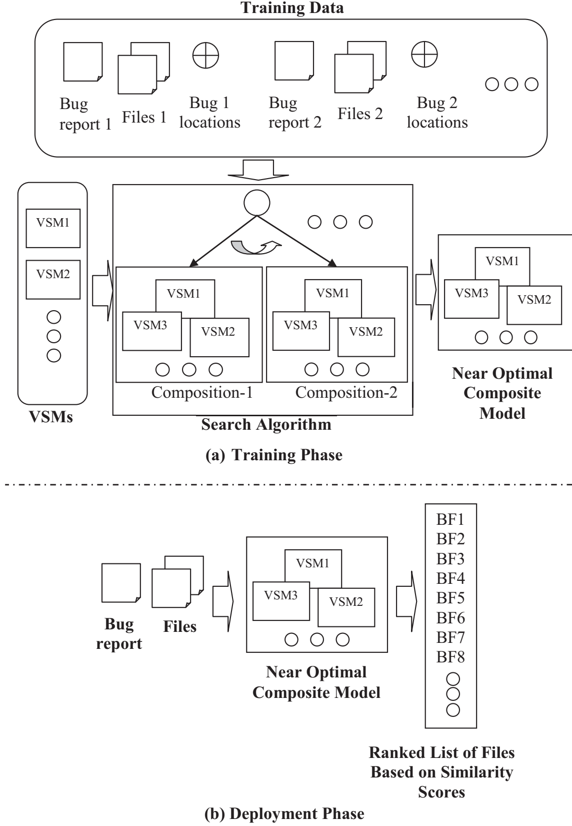

Fig. 2. Proposed Framework: Training and Deployment Phase

the bugs. In practice, this training data could be obtained from past *manual* bug localization or feature localization efforts in one or more systems. Based on the 15 VSMs and the training data, the search algorithm investigates various compositional models to find the one that best localizes bugs.

We first define the search space of potential composite models. Next, we describe how we adapt an existing search algorithm to the problem of bug localization. The training phase produces a near-optimal model, which is a composition of the 15 VSMs that performs well on the training data.

1) Search Space: There are many possible ways to combine the individual vector space models with various tf-idf weighting schemes. In this work, we create a model that computes a weighted sum of the outputs of the individual models. Given the 15 vector space models, VSM_1, VSM_2, ..., VSM_{15}, as described in Section III, a bug report b, and a file f, the similarity score of b and f is calculated by the composite model, defined as follows:

$$M_{Composite}(b,f) = \sum_{i=1}^{15} w_i \times VSM_i(b,f)$$

where w_i is the weight of the ith VSM and $VSM_i(b,f)$ is the cosine similarity score computed by vector space model VSM_i, defined in Section III. The search space of all possible compositions corresponds to the various assignments of weights, *i.e.*, $w_1, w_2, \ldots, w_{15}$. These weights are real numbers whose values range from zero to one.

2) Objective Functions: Search algorithms require an objective function to measure how good a candidate solution is. As introduced in Section II, the goal of a genetic algorithm is to maximize the value of a given objective function.

Before defining the objective function for our GA, we first introduce two evaluation metrics that are commonly used to measure the effectiveness of bug localization techniques [5], [12]–[14].

Mean Average Precision (MAP): MAP is the most commonly used IR metric to evaluate ranking approaches. It considers the ranks of all buggy files. Therefore, MAP emphasizes all of the buggy files instead of only the first one. MAP is computed by taking the mean of the *average precision* scores across all bug reports. The average precision of a bug report is computed as:

$$AP = \sum_{k=1}^{M} \frac{P(k) \times pos(k)}{\#buggy\ files},$$

where k is a rank in the returned ranked files, M is the number of ranked files and $pos(k)$ indicates whether the kth file contains the bug or not. $P(k)$ is the precision at a given top k files and is computed as follows:

$$P(k) = \frac{\#buggy\ files\ among\ top\text{-}k\ files}{k}.$$

Mean Reciprocal Rank (MRR): The reciprocal rank for a bug report is the reciprocal of the position of the first buggy file in the returned ranked files. MRR is the mean of the reciprocal ranks over a set of bug reports Q and it can be computed by the following equation:

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

where $rank_i$ is the position of the first file containing the bug in the returned ranked files for a bug report in Q.

Our goal is to maximize MAP and MRR, as the higher their values are, the more effective the composite model is. Thus we define our objective function as:

$$obj = MAP + MRR$$

We give an equal weight to MAP and MRR since both of them are important. MAP is important if a developer is interested to find all buggy files by reading the recommended files one by one. MRR is important if a developer is interested in only finding the first buggy file; based on this file, he can then start his manual debugging effort to find the other related buggy files.

3) Adapting Search Algorithm: To use a genetic algorithm for our problem, we model the search space as described in Section IV-A1. Each solution (chromosome) in the search space is represented as an array of floats ranging from 0 to 1. A genetic algorithm starts with an initial population of chromosomes (i.e., initial solutions) and iteratively produces new populations by performing selection, crossover, and mutation operations. We create the initial population randomly. We use the standard crossover and uniform mutation operations [24] and adapt only the selection operation. One-point crossover randomly swaps some genes (i.e., the values of some weights) between two randomly selected chromosomes. Uniform mutation randomly replaces one of the genes (i.e., the value of one of the weights) by a real number between zero and one.

The selection operation of our GA selects the best chromosomes, according to a *fitness function*, to be used as the population for the next generation. Considering obj to be the objective function that needs to be maximized, we define the fitness score of a chromosome X_i in a generation R containing $|R|$ chromosomes as the chromosome's relative performance against other chromosomes, as follows:

$$Fitness(X_i, R) = \sum_{j=1}^{|R|} e^{10\times(obj(X_j)-obj(X_i))}$$

As the values of $obj(X_j)$ and $obj(X_i)$ are often very similar,[1] to compute the fitness of a chromosome, we take the exponentiation of the differences between $obj(X_j)$ and $obj(X_i)$ multiplied by ten, to make the differences more significant. This difference needs to be significant so that "fitter" chromosomes have significantly more chance when chromosomes are randomly selected based on their fitness scores to create the population for the next generation.

Our selection operation first selects top chromosomes based on three criteria and then randomly selects other chromosomes based on their fitness scores. We begin with some definitions.

Definition 4.1 (AP_k^c Notation): Let AP denote the average precision. We define $APs(c) = \{AP_1^c, AP_2^c, \ldots, AP_k^c\}$ to denote the APs of the k bug reports, using the composite VSM constructed based on chromosome c. The compositional VSM uses the weights specified in chromosome c.

Definition 4.2 ($imp(c1, c2)$ Notation): To compare the accuracy of the composite VSMs generated by two chromosomes $c1$ and $c2$ in terms of the AP value for each bug report, we also define

$$imp(c1, c2) = \{\frac{AP_1^{c1} - AP_1^{c2}}{AP_1^{c1}}, \frac{AP_2^{c1} - AP_2^{c2}}{AP_2^{c1}}, \ldots, \frac{AP_k^{c1} - AP_k^{c2}}{AP_k^{c1}}\}$$

to denote the set of the improvement of $c2$ over $c1$ in terms of AP for each bug report.

Definition 4.3 ($compare_{\#wins}(c1, c2)$ Notation): Given two chromosomes $c1$ and $c2$, $compare_{\#wins}$ counts the number of elements in $imp(c1, c2)$ that are larger than 0 (referred to as *winCases*), and the number of elements in $imp(c1, c2)$ that are smaller than 0 (referred to as *loseCases*). If $winCases \geq loseCases$, return "$c2$ performs at least as well as $c1$"; otherwise, return "$c1$ beats $c2$".

Definition 4.4 ($compare_{sum}(c1, c2)$ Notation): Given two chromosomes $c1$ and $c2$, $compare_{sum}$ sums the elements in the set $imp(c1, c2)$. If the sum ≥ 0, return "$c2$ performs at least as well as $c1$"; otherwise, return "$c1$ beats $c2$".

Based on the notations defined above, we use the following criteria to select the top chromosomes. We select each chromosome c that satisfies *all* of the following selection criteria as one of the top chromosomes:

[1] Especially for the later iterations when the chromosomes in the population are getting more similar (i.e., the search space starts to converge).

1) The fitness score for c is no less than the 90% percentile of the fitness scores of all chromosomes in the generation.
2) c performs at least as well as the best chromosome found so far, c_{best}, as measured by $compare_{\#wins}$ (c_{best}, c).
3) c performs at least as well as the best chromosome found so far, c_{best}, as measured by $compare_{sum}$ (c_{best}, c).

Typically, only a few top chromosomes satisfy all of the selection criteria selected. The remaining chromosomes to be included in the next generation are selected using the roulette wheel algorithm [22]. The roulette wheel algorithm, also known as fitness proportionate selection, randomly selects chromosomes where the probability of a chromosome being selected is proportional to the chromosome's fitness score. The pseudocode of the selection operation is shown in Figure 3.

```
Procedure Selection
Inputs:   C: Current generation of chromosomes
          P: Size of a population
          BC: Best chromosome so far
Outputs:  N=Next generation of chromosomes
Method:
1:  Initialize N with empty set
2:  Let F = set of fitness scores for each chromosome in N
3:  For (i = 0; i < P ; i++)
4:      Calculate fitness score F[i] for chromosome C[i]
5:  For (i = 0; i < P ; i++)
6:      If (C[i] satisfies the selection criteria)
7:          N[i] = C[i]
8:      Else
9:          N[i] = Randomly select from C based on
            the roulette wheel algorithm
10: Output N
```

Fig. 3. Selection Algorithm: Pseudocode

A small-scale initial experiment is used to fine-tune the GA's parameters (N_C, N_I, P_M, and P_C in Figure 1). We set the values of N_C, N_I, P_M, and P_C to 50, 200, 0.01, and 0.6, respectively.

B. Deployment Phase

During deployment, the composite model learned in the training phase is used to localize new bug reports. Given a set of files and a target bug report to be localized, the model is used to assign similarity scores to the files. The files are then ranked based on their similarity scores. We illustrate this process in Figure 2.(b). The resulting ranked list of files is then output to the developers.

V. AMALGAM

The composite VSM algorithm that we have developed so far is suitable for comparing bug reports to a collection of potentially buggy files. Nevertheless, recent work on bug localization has obtained better results by taking into account multiple sources of information. In this section, we first introduce *AmaLgam* which is a state-of-art bug localization approach incorporating version history, structure, and similar bug reports [14]. We then present one strategy for combining our compositional VSM with *AmaLgam*.

A. AmaLgam

AmaLgam incorporates three components to localize bugs in systems: version history, structure, and similar bug report.

The **version history component** takes as input commit logs collected from the version control system and outputs a list of files with their suspiciousness scores. The general idea of this component is that the files that have contained more bugs in the past are more likely to be buggy in the future. The suspiciousness score $score_H(b, f)$ of file f for a given bug report b is calculated with following equation:

$$score_H(b, f) = \sum_{c \in R \wedge f \in c} \frac{1}{1 + e^{12(1-((k-t_c)/k))}}$$

where R is the set of bug fixing commits that were made in the last k days before bug report b was submitted, and t_c is the number of days that have elapsed between a commit c and bug report b. We set k to 15 which is the default setting used in previous experiments with AmaLgam [14].

The **structure component** takes as input the source code corpus and a given bug report and returns a list of files with their suspiciousness scores. It categorizes the terms in the bug report as being from the summary or description fields and categorizes the terms in the source code files as being class names, method names, variable names or comments. The suspiciousness score $score_S(f, b)$ of file f for a given bug report b is computed as:

$$score_S(b, f) = \sum_{fp \in f} \sum_{bp \in b} sim(fp, bp)$$

where fp is a category from file f, bp is a category from bug report b, and $sim(fp, bp)$ is the cosine similarity of the vector representations of fp and bp, as described in [14]. The output of the structure component is a set of suspiciousness scores, one for each file.

The **similar report component** considers historical bug reports that have already been fixed. For a given bug report, it returns a list of files related to the bug report with their suspiciousness scores. The suspiciousness score $score_R(f, b, B)$ of file f for a given bug report b considering a set of historical fixed bug reports B is computed as:

$$score_R(b, f, B) = \sum_{b' \in B} I(f, b') \times \frac{sim(b, b')}{|b'.Fix|}$$

where $I(f, b')$ is an indicator function that returns 1 if f is among the set of files that were modified to fix bug report b' and it returns 0 otherwise, $sim(b, b')$ is the similarity of bug report b and a historical fixed bug report b' in B, $b'.Fix$ is the set of files that are modified to fix bug report b', and $|b'.Fix|$ is the size of the set $b'.Fix$. $sim(b, b')$ is calculated using the procedure described in [12]. Using this equation, the more frequently a file f was changed to fix historical reports that are similar with the given report b, the higher f's suspiciousness score will be.

AmaLgam combines the three components to assign a new suspiciousness score to each file. It first merges the structure and similar report components. The resulting merged component is then merged with the history component. In

TABLE IV. DATASET DETAILS

Project	Description	Period	#Fixed Bugs	#Source Files
AspectJ	Aspect-oriented extension of Java	07/2002-10/2010	286	6485
Eclipse	Open source IDE	10/2004-03/2011	3075	12863
SWT	Open source widget toolkit	10/2004-04/2010	98	484

the merging process, a set of constants, whose values are empirically determined, are used to determine the relative importance of each component. Files in a code base are then sorted based on their final suspiciousness scores (in descending order) and output for developer inspection.

B. Compositional Model: $AmaLgam_{composite}$

We combine the VSM models with different tf-idf weighting schemes, and the above three components of *AmaLgam* as follows. Given a bug report b and a set of historical fixed bug reports B, we compute the suspiciousness score $M_{Composite}(b, f)$ of file f as follows:

$$\sum_{i=1}^{15} w_i \times VSM_i(b,f) + \sum_{J}^{J \in H,R,S} w_j \times score_J(b,f)$$

In the above equation, to compute the suspiciousness score of a file we take a weighted sum of the suspiciousness scores computed by the different VSM models and the three components of *AmaLgam*. We use a genetic algorithm following the same procedure that is presented Section IV to get a semi-optimal compositional model. In this case, each chromosome contains three more real numbers, which correspond to the weights of the three components of *AmaLgam*. We refer to this compositional model as $AmaLgam_{composite}$.

VI. EMPIRICAL EVALUATION

In this section, we first describe our experimental setting and then present our evaluation results.

A. Experimental Setting

1) Datasets: We use three datasets containing a total of 3,459 bug reports from three popular open source projects, AspectJ, Eclipse, and SWT. The details of these datasets are shown in Table IV. These bug reports were also used to evaluate *AmaLgam* [14]. In the work on *AmaLgam*, another small dataset ZXing was also used; however there are only 20 bug reports in this dataset, making it too small for our approach to learn a near-optimal compositional model. In practice, for many large systems, which benefit more from a bug localization solution, a large number of historical fixed bug reports are available, and they can be used by our approach to learn a semi-optimal compositional model.

2) Effectiveness Calculation: We use the components of our objective function, MAP (Mean Average Precision) and MRR (Mean Reciprocal Recall), defined in Section IV-A2, to evaluate the effectiveness of our solution. We also use Hit@N, as in past studies [12]–[14][2]. Hit@N is defined as follows:

[2]Hit@N is referred to as Top N in [12].

Hit@N: This metric calculates the number of bug reports where one of its buggy files appears in the top N ranked files. Given a bug report, if at least one of its relevant files is in the top N ranked files, we consider the report is successfully located (i.e., a hit). The higher the value of this metric is, the better is the performance of an bug localization technique.

3) Cross Validation: We perform a three-fold cross validation (i.e., 2/3 of the dataset is used for training and 1/3 is used for testing) [17] on each dataset to evaluate our compositional bug localization approach. We randomly split the bug reports into 3 buckets. In three-fold cross validation, three separate iterations are performed. In each iteration, we keep one of the buckets as the test set and use the other two buckets for training. Unless otherwise stated, the results reported in the following subsections are based on the average of the three iterations of the cross validation process.

B. Research Questions

RQ1 How effective is our search-based compositional approach $VSM_{composite}$ as compared to $VSM_{natural}$?

$VSM_{natural}$ is a widely used vector space model variant and it has been shown to outperform many IR-based approaches that are not based on VSM [15]. In this research question, we would like to investigate whether our search-based compositional approach is useful to improve $VSM_{natural}$. Accordingly, we evaluate $VSM_{composite}$ and $VSM_{natural}$ on three datasets. We measure the effectiveness of the approaches using Hit@1, Hit@5, Hit@10, MAP, and MRR.

RQ2 Does our compositional model $AmaLgam_{composite}$ improve $AmaLgam$?

In this research question, we would like to test whether our approach could be integrated into an existing state-of-the-art approach to improve its performance. To answer this question, we perform experiments on the three datasets and compare the effectiveness of $AmaLgam_{composite}$ and $AmaLgam$.

RQ3 How generally applicable is a compositional model trained from a set of software projects?

In this RQ, we want to test the reusability of a compositional model by investigating whether a compositional model trained on bug reports from one set of software projects can be used to localize bug reports from a different project. For this, we do a $|P|$-fold cross-project validation, where $|P|$ is now the number of software projects. Each bucket now contains bug reports from *one* software project. During each of the $|P|$ iterations of the cross-validation process, bug reports from $|P| - 1$ software projects are used as the training set, while bug reports of the remaining project are used as the test set.[3]

C. Evaluation Results

1) RQ1: $VSM_{composite}$ vs. $VSM_{natural}$: Table V presents the results for all software projects. When only the top 1 file is inspected, on average, $VSM_{composite}$ localizes 16.2% more bug reports than $VSM_{natural}$. When up to 5 and 10 files are inspected, on average $VSM_{composite}$ localizes 18.4%

[3]This is different from the cross validation procedure used in the previous research questions. For the other RQs, we perform cross validation *per dataset*; thus, the training and test sets come from the same software project.

TABLE V. PERFORMANCE COMPARISONS. $AmaL = AmaLgam$. $AmaL_{compo.}$ = $AmaLgam_{composite}$.

Project	Approach	Hit@1	Hit@5	Hit@10	MAP	MRR
AspectJ	$VSM_{natural}$	25 (8.7%)	43 (15.0%)	65 (22.3%)	0.05	0.13
	$VSM_{compo.}$	33 (11.5%)	55 (19.2%)	67 (23.4%)	0.07	0.16
	$AmaL$	127 (44.4%)	187 (65.4%)	209 (73.1%)	0.33	0.54
	$AmaL_{compo.}$	**145 (50.7%)**	**211 (73.8%)**	**227 (79.4%)**	**0.43**	**0.61**
Eclipse	$VSM_{natural}$	116 (3.8%)	456 (14.8%)	709 (23.1%)	0.01	0.01
	$VSM_{compo.}$	116 (3.8%)	544 (17.7%)	845 (27.5%)	0.01	0.01
	$AmaL$	1060 (34.5%)	1775 (57.7%)	2059 (67.0%)	0.35	0.45
	$AmaL_{compo.}$	**1108 (36.1%)**	**1905 (62.0%)**	**2187 (71.2%)**	**0.39**	**0.48**
SWT	$VSM_{natural}$	12(50.7%)	37 (73.8%)	49 (79.4%)	0.21	0.24
	$VSM_{compo.}$	14 (50.7%)	40 (73.8%)	53 (79.4%)	0.23	0.26
	$AmaL$	61 (62.2%)	80 (81.6%)	**88 (89.8%)**	0.62	**0.71**
	$AmaL_{compo.}$	**62 (63.2%)**	**83 (82.6%)**	**88 (89.8%)**	**0.63**	**0.71**

and 10.1% more bug reports respectively. In terms of MAP and MRR, on average, $VSM_{composite}$ achieves a 20.6% and 10.5% improvement over $VSM_{natural}$ respectively. We take the results for all bug reports and perform Wilcoxon signed-rank tests [25] to test whether the improvements obtained by $VSM_{composite}$ over $VSM_{natural}$ are significant. We find that the MAP and MRR improvements are significant (p-value $<$ 0.05).[4]

2) RQ2: $AmaLgam_{composite}$ vs. $AmaLgam$: Table V also compares $AmaLgam_{composite}$ and $AmaLgam$. Averaging across the three datasets, $AmaLgam_{composite}$ localizes 6.8% more bug reports than $AmaLgam$ when only the top 1 file is inspected. When the top 5 and top 10 files are inspected, averaging across the datasets, $AmaLgam_{composite}$ improves $AmaLgam$ by 8.0% and 5.0% respectively. In terms of MAP and MRR, on average, $AmaLgam_{composite}$ achieves a 14.4% and 6.5% improvement over $AmaLgam$ respectively. We take the results for all bug reports and perform Wilcoxon signed-rank tests [25] to investigate whether the improvements obtained by $AmaLgam_{composite}$ over $AmaLgam$ are significant. We find that the MAP and MRR improvements are significant.

3) RQ3: General Applicability of $AmaLgam_{composite}$: We want to investigate whether our approach outperforms a baseline considering the $|P|$-fold cross-project validation. We compare the performance of $AmaLgam_{composite}$ with $AmaLgam$ which is the best performing baseline. Aggregating across the three datasets, 1,257, 2,070, and 2,379 bugs can be localized by just inspecting the top 1, 5, and 10 files returned by $AmaLgam_{composite}$ respectively. The average MAP and MRR scores of $AmaLgam_{composite}$ are 0.46 and 0.59 respectively. On the other hand, aggregating across the three datasets, 1,248, 2,042, and 2,356 bugs can be localized by just inspecting the top 1, 5, and 10 files returned by $AmaLgam$ respectively. The average MAP and MRR scores of $AmaLgam$ are 0.43 and 0.57 respectively. Comparing the two sets of scores, $AmaLgam_{composite}$ improves the results of $AmaLgam$ by 0.7%, 1.4%, 1.0%, 7.0% and 3.5% in terms of Hit@1, Hit@5, Hit@10, MAP and MRR, respectively.

We take the results for all bug reports and perform Wilcoxon signed-rank tests [25] to test whether the improvements obtained by $AmaLgam_{composite}$ are significant. We find that the MAP and MRR improvements are significant. The results show that a compositional bug localization model trained on bug reports from one set of software projects is still effective to localize bugs from another different project.

In practice, there is much data that we can use to train a compositional bug localization engine. Developers manually locate many bug reports daily (e.g., during bug fixing), and their bug localization activities are logged in bug repositories and version control systems [26]. By constructing a database that contains a large number of bug reports and the corresponding buggy source code files, it is likely that we can improve the result further and obtain a more general compositional bug localization engine that can accurately localize bug reports in many software projects.

D. Threats to Validity and Discussion

Threats to Validity. Threats to internal validity correspond to experimental biases. Due to space limitations, we only show a comparison between $VSM_{composite}$ and one of the VSM variants (i.e., $VSM_{natural}$). We choose $VSM_{natural}$ since it is the most widely used [16], it has been used for bug localization before [5], there is no VSM variant that is consistently better that the others, and no variant outperforms $VSM_{composite}$. Another issue is that some of the links between bug reports and their bug fixing commits may be missing from the repositories. Also, similar to prior studies, we treat all files that are modified in a bug fixing commit as buggy files. We did not manually check that the links are complete or the changed files are really buggy since we experiment with thousands of bug reports. This threat is common to other bug localization studies [5], [12]–[14], [27]. Threats to external validity correspond to the generalizability of our results. In this study, we have considered thousands of bug reports from three software projects. We have also analyzed the generalizability of our composite models by performing $|P|$-fold cross-project validation to answer our third research question. In the future, we plan to reduce this threat further by experimenting with more bug reports from more projects.

Efficiency. Because multiple iterations are involved, genetic algorithms are known to be time-consuming. Indeed, in our experiments, the training phase requires up to 3 hours on a quad core Intel Core i5-2300 2.80GHz PC with 4GB main memory (using only one core). Although training takes some time, this step only needs to be performed *once* and the resulting compositional model can be used subsequently to localize many bug reports. Occasionally, if a substantial amount of new training data is obtained, *e.g.*, from new successful bug localization efforts, a new compositional model can be learned.

Learned Weights. The weights of the compositional model learned by GA could vary from one dataset to another. This is the case as GA optimizes the weights for a dataset. Section VI-C3 demonstrates, however, that even using sub-optimal weights, learned from a different dataset, the learned compositional model could improve on $AmaLgam$, which is the state-of-the-art approach [14].

Comparison with LDA_{GA} on feature location. We also compare our approach $VSM_{composite}$ with LDA_{GA} proposed by Panichella et al. [28] on two datasets, ArgoUML and jEdit, which were used to evaluate LDA_{GA} for feature location. These datasets are publicly released by Panichella et al. along

[4]We cannot perform Wilcoxon signed-rank tests for Hit@N scores since, unlike MRR and MAP, each Hit@N score is not a mean of a distribution.

with LDA_{GA} results (i.e., the position of the first relevant method for each feature).[5] We compare the performance of $VSM_{composite}$ on ArgoUML and jEdit with the LDA_{GA} results. Since Panichella et al. only release the position of the first relevant method for each feature, we can only compute Hit@N and MRR. We find that for ArgoUML, $VSM_{composite}$ improves on LDA_{GA} in terms of Hit@1, Hit@5, Hit@10, and MRR by 100.0%, 55.6%, 46.7% and 79.9% respectively. For jEdit, $VSM_{composite}$ improves on LDA_{GA} in terms of Hit@1, Hit@5, Hit@10, and MRR by 91.7%, 58.6%, 35.5%, and 46.9%, respectively.

VII. Related Work

Feature and Bug Localization. There are many studies on feature and bug localization. Feature localization typically identifies a set of methods that implement a particular feature given a textual description of the feature. Bug localization typically identifies a set of files that contain a bug given a textual description of the bug. We highlight a few of these studies below. For a complete survey of studies on feature localization please refer to the survey by Dit et al. [29].

Antoniol et al. [20] apply a vector space model and also a probabilistic model to recover the links between textual documents (e.g., manuals) and source code units. Marcus and Maletic [30] use Latent Semantic Indexing (LSI) to solve the same problem. Wang et al. [15] systematically investigate the effectiveness of ten information retrieval techniques (including LSI, VSM, and various topic models) on hundreds of feature descriptions. They find that VSM performs best. In this work, we note that there are many variants of VSM and each of them performs well on some bug reports but poorly on others. We propose a search-based technique that combines VSM variants to build a more accurate composite model.

Search-based LDA, proposed by Panichella et al. [28], uses a genetic algorithm to find a near-optimal configuration (i.e., values of alpha, beta, topic number, and iteration number) of a topic model, namely Latent Dirichlet Allocation (LDA), based on a given raw dataset. They then use the configured LDA to perform feature localization. We use VSM variants with different weighting schemes and use a GA to search for a near-optimal combination of those variants.

Lukins et al. employ a well-known topic modeling technique named Latent Dirichlet Allocation (LDA) to localize bug reports [31]. Rao and Kak extends Lukins et al.'s study by investigating a number of information retrieval techniques to localize bugs [5]. They find that simpler techniques, i.e., VSM and SUM, outperform many other more complicated solutions. Sisman and Kak analyze version history to find files that are more prone to be buggy and use this information to improve bug localization by creating a history-aware solution [27]. Zhou et al. measure similarities among bug reports and retrieve similar fixed bug reports to a new report to identify candidate buggy source code files [12]. Saha et al. improve prior bug localization techniques by considering the structure of bug reports and source code files [13]. Wang and Lo combine the work of Sisman et al., Zhou et al., and Saha et al. by putting together version history, structure, and similar bug reports, in a technique named $AmaLgam$, to achieve better performance [14]. In this work, we extend a standard variant of VSM (i.e., $VSM_{natural}$) and $AmaLgam$ into $VSM_{composite}$ and $AmaLgam_{composite}$ respectively. We demonstrate that $VSM_{composite}$ and $AmaLgam_{composite}$ significantly outperform $VSM_{natural}$ and $AmaLgam$ respectively.

A number of approaches analyze execution traces in addition to textual information to perform feature location. Poshyvanyk et al. locate features using analysis of both natural language descriptions and execution traces of relevant scenarios that simulate the features [11]. Dit et al. employ a web mining technique to analyze execution information and merge it with information retrieval to improve feature location [32]. Our approach does not leverage execution trace information since often this information is not available in bug reports [33].

Spectrum-Based Fault Localization. Numerous approaches localize the lines of code responsible for a bug given a set of correct and failing execution traces [34]–[40]. These studies ignore textual information, and focus only on execution information. However, many bug reports include no test cases, making it hard to collect many correct and failing executions.

Search-Based Software Engineering. Gold et al. use search-based techniques for concept location [41]. They map concepts (each of which is a word, e.g., "file", "write") to lines in source code, leveraging a manually constructed knowledge base mapping concepts to indicators. They use hill climbing and a genetic algorithm to search for the best assignment of concepts to segments of code, while allowing overlapping boundaries among segments. They show that their genetic algorithm performs better than hill climbing and random search on 21 COBOL II programs. Our work is orthogonal, in that we use a search-based algorithm in a different way to solve a different problem. We search through compositions of VSM models rather than best ways to segment the code. Also, for bug localization, our approach takes as input a bug report rather than a concept, which typically is a single word or a short phrase. Finally, our approach does not require a manually constructed knowledge base mapping concepts to indicators.

Li et al. use various search algorithms, including greedy search, hill climbing, and a genetic algorithm, for test case prioritization [42]. A number of algorithms have been proposed to generate test cases that satisfy various criteria for various programs [43]. Antoniol et al. apply a genetic algorithm to allocate staff to project teams and to allocate teams to work packages [44]. Mancoridis et al. use a search-based algorithm to group software modules into clusters by minimizing cohesion and maximizing coupling [45]. Wang et al. use a genetic algorithm to improve fault localization, analyzing failing and correct execution traces to locate faulty basic blocks [46]. Our work considers a different problem (i.e., localizing buggy source code files from a textual bug description) and a different technique that composes VSM variants.

VIII. Conclusion & Future Work

Many information retrieval techniques have been proposed to localize buggy source code files from a bug report. The vector space model (VSM) with the standard tf-idf weighting scheme has been shown to outperform many non-VSM IR-based techniques [15]. However there are various VSM variants with different tf-idf weighting schemes. In this paper,

[5] http://www.distat.unimol.it/reports/LDA-GA.

we build a solution that combines 15 VSMs with different tf-idf weighting schemes into an improved composite model, constructed using a genetic algorithm. We have evaluated our approach on 3,459 bug reports from AspectJ, Eclipse, and SWT and demonstrate that our approach can achieve better performance than VSM with the standard tf-idf weighting scheme ($VSM_{natural}$). Compared with $VSM_{natural}$, averaging across the 3 datasets, our approach, $VSM_{composite}$, improves $VSM_{natural}$ in terms of Hit@5, MAP, and MRR by 18.4%, 20.6%, and 10.5% respectively. We have also combined the 15 VSMs with the 3 components of $AmaLgam$, which is the state-of-the-art bug localization technique, to create $AmaLgam_{composite}$. Compared with $AmaLgam$, averaging across the 3 datasets, $AmaLgam_{composite}$ can improve $AmaLgam$ in terms of Hit@5, MAP, and MRR by 8.0%, 14.4% and 6.5% respectively.

In the future, we plan to reduce the threats to external validity by investigating additional bug reports from additional software projects. We also plan to improve the effectiveness of our approach further. One possible way is by composing additional IR techniques, e.g., BM25, LSA, etc. [16], to our compositional model. Another possible way is by adaptively fine tuning the set of weights assigned to the VSM variants for each individual project or each individual bug report.

REFERENCES

[1] T. Ball, M. Naik, and S. K. Rajamani, "From symptom to cause: Localizing errors in counterexample traces," in *POPL*, 2003.

[2] G. Canfora and A. Cimitile, *Handbook of Software Engineering and Knowledge Engineering (Vol 1).* World Scientic, 2002.

[3] T. Standish, "An essay on software reuse," *IEEE Transactions on Software Engineering*, vol. 5, no. 10, pp. 494–497, 1984.

[4] B. Cornelissen, A. Zaidman, and A. van Deursen, "A controlled experiment for program comprehension through trace visualization," *IEEE Transactions of Software Engineering*, 2011.

[5] S. Rao and A. C. Kak, "Retrieval from software libraries for bug localization: a comparative study of generic and composite text models," in *MSR*, 2011, pp. 43–52.

[6] T. Biggerstaff, B. Mitbander, and D. Webster, "The concept assignment problem in program understanding," in *ICSE*, 1993.

[7] G. Antoniol and Y.-G. Guéhéneuc, "Feature identification: A novel approach and a case study," in *ICSM*, 2006.

[8] M. Eaddy, A. Aho, G. Antoniol, and Y.-G. Guéhéneuc, "CERBERUS: Tracing requirements to source code using information retrieval, dynamic analysis, and program analysis," in *ICPC*, 2008.

[9] T. Eisenbarth, R. Koschke, and D. Simon, "Locating features in source code," in *IEEE Transactions of Software Engineering*, 2003.

[10] T. Eisenbarth and K. D. Volder, "Dynamic feature traces: Finding features in unfamiliar code," in *ICSM*, 2005.

[11] D. Poshyvanyk, Y.-G. Guéhéneuc, A. Marcus, G. Antoniol, and V. Rajlich, "Feature location using probabilistic ranking of methods based on execution scenarios and information retrieval," *IEEE Transactions of Software Engineering*, vol. 33, no. 6, pp. 420–432, 2007.

[12] 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 *ICSE*, 2012, pp. 14–24.

[13] R. K. Saha, M. Lease, S. Khurshid, and D. E. Perry, "Improving bug localization using structured information retrieval," in *ASE*, 2013.

[14] S. Wang and D. Lo, "Version history, similar report, and structure: Putting them together for improved bug localization," in *ICPC*, 2014.

[15] S. Wang, D. Lo, Z. Xing, and L. Jiang, "Concern localization using information retrieval: An empirical study on Linux kernel," in *WCRE*, 2011, pp. 92–96.

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

[17] J. Han and M. Kamber, *Data Mining Concepts and Techniques*. Morgan Kaufmann, 2006.

[18] S. Kirkpatrick, C. Gelatt, and M. Vecchi, "Optimization by simulated annealing," *Science*, vol. 220, no. 4598, pp. 671–680, May 1983.

[19] J. Holland, *Adaptation in Natural and Artificial Systems: An Introductory Analysis with Applications to Biology, Control, and Artificial Intelligence.* Univ. of Michigan, 1975.

[20] G. Antoniol, G. Canfora, G. Casazza, A. D. Lucia, and E. Merlo, "Recovering traceability links between code and documentation," *IEEE Transactions of Software Engineering*, vol. 28, no. 10, 2002.

[21] M. Porter, "An Algorithm for Suffix Stripping," *Program*, 1980.

[22] T. Bäck, *Evolutionary algorithms in theory and practice: evolution strategies, evolutionary programming, genetic algorithms*. Oxford, UK: Oxford University Press, 1996.

[23] H. Fang, T. Tao, and C. Zhai, "A formal study of information retrieval heuristics," in *SIGIR*, 2004.

[24] J. H. Holland, *Adaptation in Natural and Artificial Systems*. Cambridge, MA, USA: MIT Press, 1992.

[25] F. Wilcoxon, "Individual comparisons by ranking methods," *Biometrics Bulletin*, no. 6, pp. 80–83, 1945.

[26] T. F. Bissyandé, F. Thung, S. Wang, D. Lo, L. Jiang, and L. Réveillère, "Empirical evaluation of bug linking," in *CSMR*, 2013.

[27] B. Sisman and A. C. Kak, "Incorporating version histories in information retrieval based bug localization," in *MSR*, 2012, pp. 50–59.

[28] A. Panichella, B. Dit, R. Oliveto, M. Di Penta, D. Poshyvanyk, and A. De Lucia, "How to effectively use topic models for software engineering tasks? an approach based on genetic algorithms," in *ICSE*, 2013.

[29] 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.

[30] A. Marcus and J. I. Maletic, "Recovering documentation-to-source-code traceability links using latent semantic indexing," in *ICSE*, 2003.

[31] S. K. Lukins, N. A. Kraft, and L. H. Etzkorn, "Bug localization using latent dirichlet allocation," *Information & Software Technology*, 2010.

[32] B. Dit, M. Revelle, and D. Poshyvanyk, "Integrating information retrieval, execution and link analysis algorithms to improve feature location in software," *Empirical Software Engineering*, 2013.

[33] C. Sun, D. Lo, X. Wang, J. Jiang, and S.-C. Khoo, "A discriminative model approach for accurate duplicate bug report retrieval," in *ICSE (1)*, 2010, pp. 45–54.

[34] J. Jones and M. Harrold, "Empirical evaluation of the Tarantula automatic fault-localization technique," in *ASE*, 2005.

[35] R. Abreu, P. Zoeteweij, and A. J. C. van Gemund, "On the accuracy of spectrum-based fault localization," in *TAICPART-MUTATION*, 2007.

[36] Lucia, D. Lo, L. Jiang, and A. Budi, "Comprehensive evaluation of association measures for fault localization." in *ICSM*, 2010.

[37] Lucia, D. Lo, L. Jiang, F. Thung, and A. Budi, "Extended comprehensive study of association measures for fault localization," *Journal of Software: Evolution and Process*, vol. 26, no. 2, pp. 172–219, 2014.

[38] L. Gong, D. Lo, L. Jiang, and H. Zhang, "Interactive fault localization leveraging simple user feedback," in *ICSM*, 2012, pp. 67–76.

[39] Lucai, D. Lo, and X. Xia, "Fusing fault localizers," in *ASE*, 2014.

[40] W. E. Wong, V. Debroy, R. Gao, and Y. Li, "The DStar method for effective software fault localization," *IEEE Transactions on Reliability*, vol. 63, no. 1, pp. 290–308, 2014.

[41] N. Gold, M. Harman, Z. Li, and K. Mahdavi, "Allowing overlapping boundaries in source code using a search based approach to concept binding," in *ICSM*, 2006, pp. 310–319.

[42] Z. Li, M. Harman, and R. Hierons, "Search algorithms for regression test case prioritization," *IEEE Transactions on Software Engineering*, vol. 3, pp. 225–237, 2007.

[43] P. Tonella, "Evolutionary testing of classes." in *ISSTA*, 2004.

[44] G. Antoniol, M. D. Penta, and M. Harman, "A robust search-based approach to project management in the presence of abandonment, rework, error and uncertainty," in *METRICS*, 2004.

[45] S. Mancoridis, B. Mitchell, Y. Chen, and E. Gasner, "Bunch: A clustering tool for the recovery and maintenance of software system structures." in *ICSM*, 1999.

[46] S. Wang, D. Lo, L. Jiang, Lucia, and H. C. Lau, "Search-based fault localization," in *ASE*, 2011, pp. 556–559.