2016 IEEE International Conference on Software Maintenance and Evolution

Inferring Links between Concerns and Methods with Multi-Abstraction Vector Space Model

Yun Zhang*, David Lo†, Xin Xia*‡, Tien-Duy B. Le†, Giuseppe Scanniello‡, Jianling Sun*
*College of Computer Science and Technology, Zhejiang University, Hangzhou, China
†School of Information Systems, Singapore Management University, Singapore
‡University of Basilicata, Potenza, Italy
yunzhang28@zju.edu.cn, davidlo@smu.edu.sg, xxkidd@zju.edu.cn, btdle.2012@smu.edu.sg,
giuseppe.scanniello@unibas.it, sunjl@zju.edu.cn

***Abstract*—Concern localization refers to the process of locating code units that match a particular textual description. It takes as input textual documents such as bug reports and feature requests and outputs a list of candidate code units that are relevant to the bug reports or feature requests. Many information retrieval (IR) based concern localization techniques have been proposed in the literature. These techniques typically represent code units and textual descriptions as a bag of tokens at *one level of abstraction*, e.g., each token is a word, or each token is a topic. In this work, we propose a *multi-abstraction* concern localization technique named** MULAB**.** MULAB **represents a code unit and a textual description at multiple abstraction levels. Similarity of a textual description and a code unit is now made by considering all these abstraction levels. We combine a vector space model and multiple topic models to compute the similarity and apply a genetic algorithm to infer semi-optimal topic model configurations. We have evaluated our solution on 136 concerns from 8 open source Java software systems. The experimental results show that** MULAB **outperforms the state-of-art baseline** PR**, which is proposed by Scanniello et al. in terms of effectiveness and rank.**

***Index Terms*—Concern Localization, Multi-Abstraction, Text Retrieval, Topic Modeling**

I. Introduction

Developers receive bug reports and feature requests through issue management systems such as Bugzilla and JIRA daily. The amount of these reports are often too many for developers to handle [1]. For each of these reports and requests, developers need to locate the code units that need to be modified to fix bugs or be extended to implement a particular feature. Considering a large code base with thousands or even millions of files, this task is a daunting one. Much manual effort needs to be spent to locate relevant code units. Thus, an automated solution is needed.

Concern localization is a software maintenance process of locating code units that need to be changed in response to a modification request, such as bug fixing or a new feature request. Change requests are usually formulated in natural language, describing the problems or the solutions of the software system, while the source code also includes large amounts of text such as comments and identifiers.

Recently, a number of approaches have been proposed to link bug reports and feature requests to the corresponding code units, e.g., [2], [3], [4], [5], [6], [7], [8]. The bug reports and feature requests could be viewed as *concerns*,[1] and the linking process of code units to concerns is referred to as *concern localization*. Many past studies on bug localization, feature location, etc. could be viewed as specific instances of concern localization.

Many existing studies characterize both concerns (e.g., feature requests or bug reports) and code units as a bag (i.e., multi-set) of tokens at *one abstraction level*, e.g., [2], [3]. A textual document (i.e., feature request, bug report, or code unit) is represented as a set of words that appear in it. Alternatively, a natural language processing technique referred to as topic modeling (e.g., [10]) can be applied to infer a set of topics that appear in the document. A topic is a distribution of words and is a higher level abstraction of the original words. A set of topics can be inferred from documents and these topics would represent these documents. Similarities of documents can then be measured as the similarities of their representations (i.e., their set of words or topics). The code units that are most similar to the input concerns are output to the end user.

Recently, Scanniello et al. propose a static concern localization approach named PR which combines textual and structural information together [11]. PR extracts dependency among methods in a code base (based on direct references between methods) and uses the PageRank algorithm to rank methods based on their importance. Similarities between a concern and a code unit (i.e., a method) is then measured by multiplying the textual similarity computed by comparing the concern and the code unit using vector space modeling (VSM) and the importance of the code unit estimated using PageRank. The experiment results show that their approach leads to better retrieval performance than several baseline approaches: one that uses textual information only and one that combines textual and structural information via clustering [12].

While many past studies only compare two documents at one abstraction level, in this work, we compare documents at multiple abstraction levels. A word can be abstracted at multiple levels of abstraction. For example, Raleigh can be

‡Corresponding author.

[1] A concern is a concept, requirement, feature, or property related to a software system [9]. In this work, we focus on bug reports and feature requests which are subsets of concerns, but the proposed approach could be used for generic concerns.

978-1-5090-3806-0/16 $31.00 © 2016 IEEE
DOI 10.1109/ICSME.2016.51

abstracted to North Carolina, South Atlantic, U.S.A., North America, Earth, and so on. Two documents might not share the same word "Raleigh" but they might be about the same state (i.e., North Carolina), the same country (i.e., U.S.A.), the same continent (i.e., North America), and so on. By viewing a document at multiple levels of abstractions the similarity or difference of two documents can be better assessed.

To represent documents in multiple abstraction levels, we leverage topic modeling. Topic modeling maps words that appear in a document to topics. Each word is assigned to one topic. The fewer the number of topics, the higher the abstraction level. This is the case as a topic now represents more words. On the other hand, the larger the number of topics, the lower the abstraction level. Thus, we can iteratively apply topic modeling using different numbers of topics to create multiple abstraction levels. We can then aggregate these abstractions to measure the similarity between a concern (e.g., a bug report or a feature request) and a code unit. We apply an adaptive Latent Dirichlet Allocation (LDA) with Genetic Algorithm (GA) to determine a near-optimal configuration for LDA to tune the topic number of each abstraction level [13].

In the literature, vector space modeling (VSM) has been shown to outperform many other information retrieval (IR)-based techniques for concern localization [3], [14]. In this paper, we extend VSM to consider multi abstraction levels. We refer to the resultant model as MULti-ABstraction VSM (MULAB). We evaluate MULAB on 8 open-source software systems using information from 136 past change requests which map to a total of 388 changed methods. To demonstrate that the proposed multi-abstraction concept works, we compare MULAB with PR, the state-of-the-art proposed by Scanniello et al. [11].

This paper extends our preliminary study which appears as an ERA track paper[2] of ICSM 2013 [15]. In particular, we extend our preliminary work in several directions: *(i)* Rather than arbitrarily setting the number of topics for each abstraction level, we use LDA-GA to better tune the topic numbers; *(ii)* We strengthen the experimental part by utilizing concerns from 8 Java software systems to evaluate our technique; *(iii)* We have compared the effectiveness of our multi-abstraction approach against a recently proposed state-of-the-art approach [11].

Our contributions, which form a super-set of those of our preliminary study, are as follows:

1) We propose multi-abstraction concern localization. We represent a document (i.e., a code unit, bug report, or feature request) at multiple abstraction levels.
2) We propose a technique MULAB that leverages multiple topic models to capture representations of documents at different abstraction levels. MULAB employs an adaptive LDA with genetic algorithm (LDA-GA) to tune the topic numbers of each abstraction level. MULAB then uses these representations to compute the similarity between a concern and a code unit.
3) We have evaluated MULAB on hundreds of concerns from 8 Java software systems. Results show that our proposed multi-abstraction approach outperforms PR by a substantial margin.

Paper structure. In Section II, we briefly introduce LDA and GA. In Section III, we present the details of MULAB. We present our experimental results in Section IV. We review related work in Section V. We conclude and mention future work in Section VI.

II. Preliminaries

A. Latent Dirichlet Allocation

A topic model views a document to be a probability distribution of topics, while a topic is a probability distribution of words. In our setting, a document is a program method in the code base or a concern, and a topic is a higher-level concept corresponding to a distribution of words. For example, we can have a topic "Java Programming" which is a distribution of words such as "variable", "inheritance", "class", "method", and so on.

Latent Dirichlet Allocation (LDA) is a well-known topic modeling technique proposed by Blei et al. [16], which has been widely used in software engineering [13], [17], [18]. LDA takes a document-by-term matrix D as input, and outputs two matrices DT and TT, i.e., a document-by-topic matrix and a topic-by-term matrix. The document-by-term matrix D is a term frequency matrix, in which D_{ij} represents the number of times that the j-th term (i.e., word) appears in the i-th document. In the document-by-topic matrix DT, DT_{ij} represents the probability of the i-th document to belong to the j-th topic. Generally, a document is considered to belong to the topic with the highest probability. In the topic-by-term matrix TT, TT_{ij} represents the probability that the j-th term belongs to the i-th topic. Likewise, we assign a term to the topic with the highest probability and then we can conclude what a topic is about by looking up the terms it contains. After training, LDA can be used to infer the topic distribution of a new document (in our case: a new concern). LDA takes several parameters: the number of topics (K), and two hyper-parameters α and β. While the hyper-parameters are typically set to be $50/K$ and 0.01 respectively following the suggestions by Blei et al. [16], the values of K needs to be carefully tuned.

There are several implementations for LDA in the literature. In our work, we use an implementation based on collapsed Gibbs sampling. This approach typically achieves the same accuracy as the standard LDA implementation while being faster in its execution [19], [20]. Besides the three parameters, K, α, and β introduced above, our Gibbs sampling implementation takes an additional parameter m which specifies the number of Gibbs sampling iterations. By default, we set m to be 1,000.

B. Genetic Algorithms

A genetic algorithm (GA) is a stochastic search technique that mimics the process of natural selection. Since its first

[2] Two of the three authors of the preliminary study paper are co-authors of this paper.

introduction by Holland [21] in the 1970s, genetic algorithms have been widely used to generate solutions to optimization problems using techniques such as mutation, selection, and crossover. The advantage of GA with respect to other search algorithms is its intrinsic parallelism, i.e., having multiple solutions evolving in parallel to explore different parts of the search space.

The GA search starts with a population of randomly generated individuals, where each individual (i.e., a chromosome) represents a random parameter configuration of the optimization problem. Generally, the evolution of the whole population is an iterative process, in which each iteration is called a generation. In particular, the population evolves through subsequent generations and, during each generation, the individuals are evaluated based on a fitness function that has to be optimized. The fitness function is used to evaluate the different parameter configurations by generating different fitness values. For creating the next generation, new individuals (i.e., offsprings) are generated by: (1) applying a selection operator, which randomly picks individuals based on the fitness function (individuals with higher fitness values are more likely to be selected), (2) recombining, with a given probability, two individuals from the current generation using the crossover operator, and (3) modifying, with a given probability, individuals using the mutation operator. The new generation of candidate solutions is then used in the next iteration of the algorithm. Commonly, the algorithm terminates when either a maximum number of generations have been produced, or a satisfactory fitness level has been reached for the population. More details about GA can be found in a book by Goldberg [22].

III. MULAB

A. Overview

Figure 1 presents the overall framework of MULAB. Our framework takes as input *method corpus* and *concerns*. *Method corpus* is a collection of textual documents where each document corresponds to a method in the code base. Each document contains identifiers and words that appear in the source code, documentation (e.g., Javadoc), and implementation comments of the corresponding methods. *Concerns* are a collection of textual documents where each document is either a bug report or a feature request. For each bug report and feature request, we extract the text that appears in its title and description. The output of our framework is a set of ranked methods for each concern.

Our framework contains four processing steps: *preprocessing*, *topic number tuning*, *hierarchy creation*, and *multi-abstraction retrieval*. The purpose of the *preprocessing* step is to convert methods and concern documents into a standard representation, i.e., a bag of words. The preprocessed documents (i.e., methods and concerns) are then input to the *topic number tuning* step. The *topic number tuning* step uses a genetic algorithm to determine a near-optimal topic number of LDA for each abstraction level and these are input to the *hierarchy creation* step. The *hierarchy creation* step applies a topic modeling technique a number of times to construct an *abstraction hierarchy*. The *abstraction hierarchy* is a collection of topic models with various settings, where each topic model is a level in the hierarchy. This *abstraction hierarchy* is used by the *multi-abstraction retrieval* step. In this step, we enhance a *standard text retrieval technique* based on vector space modeling (VSM) by leveraging the *abstraction hierarchy*. The goal of the final processing step is to compare a concern (a query) and a method (a document in the *method corpus*) by considering multiple abstraction levels. We elaborate the four processing steps in the following subsections.

B. Preprocessing Step

We first perform *text normalization* by removing common Java keywords (e.g., *public*, *private*, *class*, *extends*, etc.), and English stopwords. These words are deemed useless for retrieving relevant code units (i.e., methods) for concerns as either they appear in most documents or they carry little meaning. We also normalize the text by excluding punctuation marks and special symbols. Thus, we only retain some word tokens and number literals. Furthermore, we break identifiers into smaller tokens following Camel casing convention that is the naming convention adopted by most Java programs. By performing text normalization, we standardize word tokens in *Method Corpus* with those that are used in *Concerns*.

Next, we apply the Porter Stemming Algorithm[3] to reduce English words into their root forms. For example, "models", "modeled", "modeling" are all reduced to the same root word "model". We perform this step to standardize words of the same meaning but are in different forms. At the end of this step, we forward preprocessed method and concern documents to the *topic number tuning* step to determine best settings to infer topic models.

C. Topic Number Tuning Step

The parameter K of LDA, which is the number of topics, is an important parameter that significantly determines LDA output. An improper value of K for each abstraction level may affect the performance of our approach. Therefore, we use an adaptive LDA technique, leveraging genetic algorithm (GA), to optimize the value of K in each abstraction level. This approach proposed by Panichella et al. is referred to as LDA-GA [13].

At the beginning, a population of p randomly-generated chromosomes is initialized by LDA-GA, where each of chromosome contains a random integer value corresponding to the number of topics. Then, the population will evolve in n generations to search for an optimal value of the number of topics. The population is evolved relying on a fitness function which corresponds to the Silhouette coefficient. The Silhouette coefficient is a common evaluation metric for measuring the goodness of a clustering result [13], [23], [24], [25]. In LDA-GA, documents are clustered according to the topics inferred by LDA, where documents assigned to the same topic are

[3] http://tartarus.org/martin/PorterStemmer/

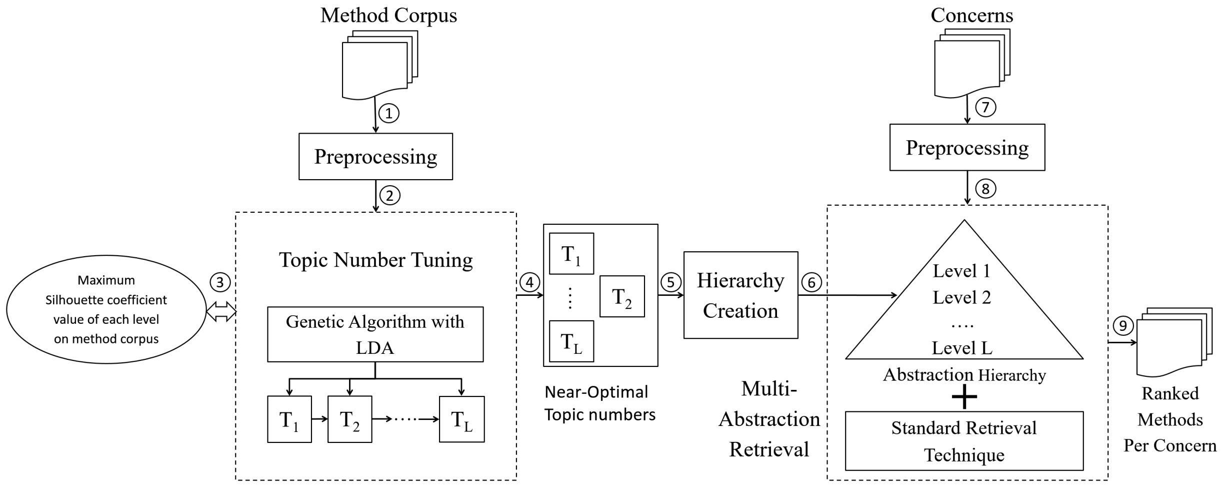
Method Corpus
①
Preprocessing
②
Topic Number Tuning
Maximum Silhouette coefficient value of each level on method corpus
③
Genetic Algorithm with LDA
T_1
T_2
T_L
④
Near-Optimal Topic numbers
⑤
Hierarchy Creation
⑥
Multi-Abstraction Retrieval
Concerns
⑦
Preprocessing
⑧
Level 1
Level 2
....
Level L
Abstraction Hierarchy
Standard Retrieval Technique
⑨
Ranked Methods Per Concern

Fig. 1. Overall Framework of MULAB

grouped in the same cluster. From these clusters of documents, three steps are performed to compute the Silhouette coefficient:

1) **Step 1:** For a document d_i, we calculate the maximum distance from d_i to the other documents in the same cluster, which is denoted as a(d_i). And we calculate the minimum distance from d_i to the centroids of the other clusters not containing d_i, which is denoted as b(d_i).

2) **Step 2:** Given $a(d_i)$ and $b(d_i)$, we can calculate the Silhouette coefficient $s(d_i)$ for the document d_i according to the following formula:

$$s(d_i) = \frac{b(d_i) - a(d_i)}{max\{a(d_i), b(d_i)\}}$$

3) **Step 3:** We compute the mean value of all s(d_i) as the overall Silhouette coefficient.

The range of the Silhouette coefficient is $[-1, 1]$. A larger value of the Silhouette coefficient indicates a better clustering. When a high Silhouette coefficient is achieved for a particular value of the number-of-topic parameter of LDA, it means that the particular parameter value leads to a good result. The higher Silhouette coefficient is achieved using a particular parameter value, the more likely the parameter value is kept in the genetic algorithm (GA)'s evolution process. Therefore, by using LDA-GA, we can find a suitable number of topics for each abstraction level.

The original implementation of LDA-GA is written in R and it runs rather slowly. Thus, we reimplement LDA-GA approach on the top of *Pyevolve*,[4] an evolutionary computation framework. By default, we set p as 100 and n as 50 and *Pyevolve*'s crossover and mutation rate to be 0.09 and 0.02, respectively. For each abstraction level, we execute LDA-GA to generate an optimal value of number of topics. We set different search ranges for each abstraction level. For example, let us assume that there are L levels in an abstraction hierarchy. For the first level, we set the search range to be integers in the interval $[2, \frac{D}{L}]$ where D refers to the total number of documents in the data set. We set the search range as such since we assume there should be at least 2 and at most D topics (i.e., each document belongs to its own topic). Let us assume that we a get an optimal result t_1 for this range. For the second level, we set the range to be integers in $[t_1, \frac{2D}{L}]$ and get the optimal number of topics t_2. The process repeats for the subsequent levels. Finally, for the L^{th} level, we set the search range in $[t_{L-1}, D]$, and get the best number of topics t_L for this last level. This set of L topic numbers is then output to the hierarchy creation step.

D. Hierarchy Creation Step

In the hierarchy creation step, we apply LDA a number of times to create the abstraction hierarchies with the number of topics inferred by the *topic number tuning* step. These L abstraction levels form an abstraction hierarchy H. Topic models with fewer topics are higher in the hierarchy while those with more topics are lower in the hierarchy. We refer to the number of topic models contained in a hierarchy as the *height* of the hierarchy. At the end of this step, we create an abstraction hierarchy which is used in the next step: *multi-abstraction retrieval*.

E. Multi-Abstraction Retrieval

In this subsection, we discuss how to combine an abstraction hierarchy with a text retrieval model (i.e., VSM). A retrieval method takes a query (i.e., a bug report) and returns a sorted list of most similar documents in a corpus (i.e., methods).

In standard VSM, a document is represented as a vector of weights. Each element in a vector corresponds to a word, and its value is the weight of the word. Term frequency-inverse document frequency ($tf-idf$) is often used to assign weights to words [26]. The following is the $tf-idf$ weight of word

[4] http://pyevolve.sourceforge.net/

w in document d given a corpus (i.e., a set of documents) D, denoted as $\textit{tf-idf}(w, d, D)$:

$$\textit{tf-idf}(w, d, D) = \log(f(w, d) + 1) \times \log \frac{|D|}{|\{d_i \in D | w \in d_i\}|}$$

where $f(w, d)$ is the number of times word w appears in document d, and $w \in d_i$ denotes that word w appears in document d_i. Given a query document q, standard VSM retrieval model would return the most similar documents in the corpus D. Similarity between two documents is measured by computing the cosine similarity between the two documents' vector representations [26].

In MULAB, we integrate abstraction hierarchy into standard VSM by extending the vector that represents a document. We added more elements to the vector. Each added element corresponds to a topic of a topic model in the abstraction hierarchy, and its value is the probability of the topic to appear in the document. The size of an extended document vector is $V + \sum_{i=1}^{L} K(H_i)$, where V is the size of the original document vector, L is the number of abstraction levels in the hierarchy, and $K(H_i)$ is the number of topics of the i^{th} topic model in the abstraction hierarchy H. Based on this representation, the similarity between a query q and document d, considering a corpus D, calculated using cosine similarity, is as follows:

$$sim(q, d, D) = \frac{\sum_{i=1}^{V} \textit{tf-idf}(w_i, q, D) \times \textit{tf-idf}(w_i, d, D) + \sum_{k=1}^{L} \sum_{i=1}^{K(H_k)} \theta_{q,t_i}^{H_k} \times \theta_{d,t_i}^{H_k}}{\|q\| \times \|d\|}$$

where

$$\|q\| = \sqrt{\sum_{i=1}^{V} \textit{tf-idf}(w_i, q, D)^2 + \sum_{k=1}^{L} \sum_{i=1}^{K(H_k)} (\theta_{q,t_i}^{H_k})^2}$$

and

$$\|d\| = \sqrt{\sum_{i=1}^{V} \textit{tf-idf}(w_i, d, D)^2 + \sum_{k=1}^{L} \sum_{i=1}^{K(H_k)} (\theta_{d,t_i}^{H_k})^2}$$

In the above equations, $\theta_{d,t_i}^{H_k}$ is the probability of topic t_i to appear in the document d as assigned by the k^{th} topic model in the abstraction hierarchy H.

For example, assuming that a bug report *br* after text preprocessing has the following 7 words: *"source"*(3), *"control"*(2), *"activity"*(2), *"reduce"*(2), *"tool"*(1), *"root"*(1), *"list"*(1). We also have two methods m_1 and m_2. Each of them contains 5 words: m_1 ={*"source"(7), "control"(4), "activity"(3), "root"(7), "list"(1)*} and m_2 ={*"source"(10), "control"(10), "reduce"(5), "tool"(4), "root"(6)*}. The number in parentheses is the number of times a word appears in a document. Let us assume that an abstraction hierarchy of height 1 is used, and the topic model has 3 topics. Let us also assume that there are 1000 methods, and terms in m_1 and m_2 do not appear in other methods. Considering only the 7 words, the representative vectors of *br*, m_1, and m_2 are:

$$V_{br} = [1.62, 1.29, 1.43, 1.43, 0.90, 0.81, 0.90, \mathit{0.26}, \mathit{0.72}, \mathit{0.02}]$$
$$V_{m_1} = [2.44, 1.89, 1.81, 0.00, 0.00, 2.44, 0.90, \mathit{0.00}, \mathit{0.99}, \mathit{0.00}]$$
$$V_{m_2} = [2.81, 2.81, 0.00, 2.33, 2.10, 2.28, 0.00, \mathit{0.57}, \mathit{0.43}, \mathit{0.00}]$$

The first 7 entries in each vector are the weights of the 7 words computed using the $tf - idf$ formula, and the last 3 entries are the rounded probabilities $\theta_{d,t_i}^{H_1}$ of topics 1, 2 and 3 respectively in the documents. Finally, we calculate cosine similarities between bug report *br* and methods m_1 and m_2. The results are $sim(br, m_1) = 0.82$ and $sim(br, m_2) = 0.84$. Thus, m_2 is more relevant to bug report *br* than m_1.

IV. Experiment and Analyses

In this section, we evaluate the effectiveness of MULAB and compare it with other approaches. The experimental environment is an Intel(R) Core(TM) i7-4710HQ 2.50 GHz CPU, 16GB RAM desktop running Windows 10 (64-bit).

A. Dataset

We use datasets from 8 open source Java software systems (including two versions of one of these systems, namely jEdit) for our experimentation. In the datasets, there are totally 136 concerns which map to 388 methods. The Java systems are the same as those used by Scanniello et al. [11] except the Eclipse 3.5 system for which we do not have the source code files. In our experiment, the content of a concern is the textual description retrieved from the title and description of a bug report or a change request, along with methods that are relevant to it, i.e., methods modified to address the concern.

Each Java method is treated as a document, and all of the Java methods form a corpus. Table I shows the statistics of the data sets used in the experiment after preprocessing. The first column shows the names of the software systems and the URLs of their official web pages. The analyzed version of each system and the number of classes are reported in the second and third columns, respectively. The total number of methods in each system is shown in the fourth column, while the fifth column presents the number of concerns used in the study. The number of relevant methods is shown in the sixth column. A short description of each system is presented in the last column.

B. Evaluation Metrics

Concern localization takes a bug report and a collection of methods as input, and returns a ranked list of these methods. We use two performance metrics to evaluate a concern localization solution: *effectiveness* and *rank*, which are commonly used for concern localization studies [11], [12], [27], [28] and used to evaluate our baseline approach *PR*.

Effectiveness refers to the position of the first relevant method in the returned ranked list. Once such a method is reached, developers can determine what other methods need to be changed by analyzing the relationships between the methods. *Rank* refers to the sum of the positions of the relevant

TABLE I
DATASET

System	Version	Classes	Methods	Concerns	Changed Methods	Description
Art of Illusion (www.artofillusion.org)	2.4.1	453	6,229	8	12	A free, open source 3D modeling and rendering studio
aTunes (www.atunes.org)	1.10	419	3,712	16	30	A full featured audio player and manager.
jEdit (www.jedit.org)	4.2	411	5,384	16	33	A text editor for programming with an extensible plug-in architecture.
jEdit (www.jedit.org)	4.3	492	7,095	4	9	A text editor for programming with an extensible plug-in architecture.
Cocoon (cocoon.apache.org)	2.2	833	5,612	14	38	A spring-based framework built on separation of concerns and component-based development.
Derby (db.apache.org/)	10.7.1.1	3,418	40,278	29	80	A pure Java relational database engine of using standard SQL and JDBC as its APIs.
Lucene (lucene.apache.org)	4.0	5,199	24,682	24	112	A full-featured text search engine library.
OpenJPA (openjpa.apache.org)	2.0.1	4,765	41,474	25	74	An open source implementation of the Java Persistence API specification.

methods in the returned ranked list. *Effectiveness* and *rank* nicely complement each other; in fact, effectiveness gives us a best case scenario when an ideal user is performing a concern localization task. Conversely, rank indicates the total effort needed to identify all relevant methods for a given concern by following the ranked list (i.e., a worst case scenario). The lower the effectiveness and rank values, the better a concern localization technique is.

C. Research Questions

Research Question 1: How effective is MULAB**?**

Motivation. We investigate the effectiveness of MULAB and compare its results with those by Scanniello et al. [11] (*PR*, from here on). Answer to this research question would shed light to whether and to what extent MULAB improves over the state-of-the-art approach.

Approach. To answer this research question, we report the results obtained by applying MULAB and PR to our dataset mentioned in Section IV-A. MULAB takes in one parameter L which is the height of the abstraction hierarchy. For this RQ, we set L to be 4. We compute the effectiveness and rank scores of MULAB and PR for each concern and calculate the number of concerns for which each of the approach outperforms (or achieves the same scores as) the other.

To check if the differences in the performance of MULAB and PR are statistically significant, we apply the Wilcoxon signed-rank test [29] at 95% significance level on two paired data of all the 136 concerns which corresponds to the effectiveness and rank scores of two competing approaches respectively. We do not apply the test to each system as the numbers of concerns in some systems are small(e.g., 4 for jEdit4.3, 8 for Art of Illusion), it makes no sense to do the statistical test.

We also use Cliffs delta (δ) [30], which is a non-parametric effect size measure that quantifies the amount of difference between two approaches. In our context, we use Cliffs delta to compare MULAB with PR. The delta values range from -1 to 1, where $\delta = -1$ or 1 indicates the absence of overlap between two approaches (i.e., all values of one group are higher than the values of the other group, and vice versa), while $\delta = 0$ indicates the two approaches are completely overlapping. Table II describes the meaning of different Cliffs delta values and their corresponding interpretation [30].

TABLE II
CLIFFS DELTA AND THE EFFECTIVENESS LEVEL [30]

Cliffs Delta($\|\delta\|$)	Effectiveness Level
$\|\delta\| < 0.147$	Negligible
$0.147 \leq \|\delta\| < 0.33$	Small
$0.33 \leq \|\delta\| < 0.474$	Medium
$\|\delta\| \geq 0.474$	Large

Results. Table III presents the analysis results of effectiveness scores of MULAB and PR. The second column represents the number of concerns on which MULAB achieves better effectiveness scores than PR, the third column indicates the number of concerns on which PR performs better than MULAB, and the last column shows the number of concerns on which MULAB and PR achieve the same scores. We also report the overall results of the 8 systems in the last row. The results demonstrate that MULAB is more effective than PR on all but one of the Java systems. For one of the Java systems (i.e., Art of Illusion), both perform equally well. Among the 136 concerns, MULAB performs better on 90 concerns, PR performs better on 41 concerns, and the two approaches achieve the same effectiveness scores on 5 concerns. We have also performed a Wilcoxon signed-rank test and found that the difference in the effectiveness scores is significant with a p-value of < 0.001. The Cliff's delta is 0.348, which corresponds to a medium effect size.

Table IV presents the analysis results of rank scores of MULAB as compared with those of PR. The second column represents the number of concerns on which MULAB achieves better rank scores than PR, the third column indicates the number of concerns on which PR performs better than

TABLE III

Data analysis results on effectiveness scores of MULAB and PR. #Wins = Number of concerns for which MULAB outperforms PR, #Loses = Number of concerns for which MULAB loses from PR, #Draws = Number of concerns for which both approaches achieve the same effectiveness scores.

Systems	#Wins	#Loses	#Draws
Art of Illusion	4	4	0
aTunes	9	6	1
jEdit4.2	9	7	0
jEdit4.3	3	1	0
Cocoon	8	2	4
Derby	24	5	0
Lucene	19	5	0
OpenJPA	14	11	0
Overall	**90**	**41**	**5**

TABLE IV

Data analysis results on Rank scores of MULAB and PR. #Wins = Number of methods for which MULAB outperforms PR, #Loses = Number of methods for which MULAB loses from PR, #Draws = Number of methods for which both approaches achieve the same rank scores.

Systems	#Wins	#Loses	#Draws
Art of Illusion	6	6	0
aTunes	19	10	1
jEdit4.2	21	12	0
jEdit4.3	7	2	0
Cocoon	26	6	6
Derby	59	21	0
Lucene	81	31	0
OpenJPA	41	33	0
Overall	**260**	**121**	**7**

MULAB, and the last column shows the number of concerns on which MULAB and PR achieve the same scores. We also report the overall results of the 8 systems in the last row. From the table, we can see that for the 8 systems, MULAB performs better than that of the PR. The results demonstrate that MULAB outperforms PR on all but one of the Java systems. For one of the Java systems (i.e., Art of Illusion), both perform equally well. Among the overall 388 methods, MULAB performs better on 260 methods, PR performs better on 121 methods, and the two approaches achieve the same rank scores on 7 methods. A Wilcoxon signed-rank test shows that the difference in the rank scores is significant with a p-value of < 0.001. The Cliff's delta is 0.362, which corresponds to a medium effect size.

MULAB *outperforms* PR *on all the 8 Java systems when evaluated in terms of effectiveness and rank. Statistical tests show that the differences are statistically significant and substantial.*

Research Question 2: What is the effect of varying the text used to represent a concern on MULAB's effectiveness?

Motivation. By default, we use the text in the summary and description fields of bug reports and change requests to represent a concern – which is the setting used for RQ1 and RQ3. In this research question, we investigate the performance of MULAB when we only use text in the summary field and text in the description field independently. We want to investigate if our default setting is a better option.

Approach. To answer this research question, we conduct an experiment with three kinds of text to represent a concern: default (summary and description), summary only, description only. MULAB takes in one parameter L which is the height of the abstraction hierarchy. For this RQ, we set L to be 4. We compare the effectiveness and rank scores achieved by MULAB using each of the three kinds of text.

Results. The experiment results are shown in Tables V and VI. For each kind of text (default, summary, or description), we report the number of wins, loses, and draws for each Java system. Wins, loses, and draws represent the number of concerns[5] or methods[6] for which a particular kind of text performs the best, performs worse than another, and performs as well as the others, respectively. We also report the overall results in the last row.

Table V shows the data analysis results on effectiveness scores of MULAB using the three kinds of text to represent a concern. From the table, we can see that the default setting outperforms the others. Among the 136 concerns, *default* performs the best on 66 concerns, *summary* performs the best on 46 concerns, and *description* performs the best on 48 concerns. The effectiveness scores are the same for 6 concerns. So we can draw the conclusion that our default configuration (i.e., use both summary and description) outperforms the other two in terms of effectiveness.

Table VI shows the data analysis results on rank scores of MULAB using the three kinds of text to represent a concern. From the table, we can see that the default setting outperforms the others. Among the 388 methods, *default* performs the best on 182 methods, *summary* performs the best on 116 methods, and *description* performs the best on 115 methods. The rank scores are the same for 12 methods. So we can draw the conclusion that our default configuration (i.e., use both summary and description) outperforms the other two in terms of rank.

MULAB *with default configuration (which uses text from both summary and description fields to represent a concern) performs better than when only text from summary and text from description are used independently, in terms of both effectiveness and rank scores.*

Research Question 3: What is the effect of varying the height of the topic model hierarchy on MULAB's effectiveness?

Motivation. By default, we set the height of the hierarchy L as 4, which is the setting used for RQ1 and RQ2. In this research question, we investigate the performance of MULAB for different values of L.

Approach. To answer this question, we conduct an experiment with four different hierarchy heights (i.e., L = 1, 2, 3, and 4). We then compare the results achieved by MULAB using these

[5] When effectiveness is used as a yardstick

[6] When rank is used as a yardstick

TABLE V

Data analysis results on effectiveness scores of MULAB with different kinds of text to represent a concern. #Wins = Number of concerns for which a kind of text performs the best, #Loses = Number of concerns for which a kind of text performs worse than another, #Draws = Number of concerns for which all kinds of text lead to the same score.

	Default			Summary			Description		
System	**#Wins**	**#Loses**	**#Draws**	**#Wins**	**#Loses**	**#Draws**	**#Wins**	**#Loses**	**#Draws**
Art of Illusion	3	5	0	4	4	0	3	5	0
aTunes	8	8	0	4	12	0	6	10	0
jEdit4.2	9	6	1	2	13	1	5	10	1
jEdit4.3	2	2	0	2	2	0	2	2	0
Cocoon	9	2	3	3	8	3	5	6	3
Derby	11	18	0	15	14	0	8	21	0
Lucene	13	9	2	8	14	2	9	13	2
OpenJPA	11	14	0	8	17	0	10	15	0
Overall	**66**	**64**	**6**	**46**	**84**	**6**	**48**	**82**	**6**

TABLE VI

Data analysis results on rank scores of MULAB with different kinds of text to represent a concern. #Wins = Number of concerns for which a kind of text performs the best, #Loses = Number of concerns for which a kind of text performs worse than another, #Draws = Number of concerns for which all kinds of text lead to the same score.

	Default			Summary			Description		
System	**#Wins**	**#Loses**	**#Draws**	**#Wins**	**#Loses**	**#Draws**	**#Wins**	**#Loses**	**#Draws**
Art of Illusion	5	7	0	6	6	0	3	9	0
aTunes	15	15	0	7	23	0	10	20	0
jEdit4.2	18	14	1	6	26	1	9	23	1
jEdit4.3	6	3	0	3	6	0	2	7	0
Cocoon	23	11	4	6	28	4	14	20	4
Derby	30	50	0	35	45	0	19	61	0
Lucene	53	52	7	32	73	7	31	74	7
OpenJPA	32	42	0	21	53	0	27	47	0
Overall	**182**	**194**	**12**	**116**	**260**	**12**	**115**	**261**	**12**

different hierarchy heights in terms of effectiveness and rank scores.

Results. The experiment results are shown in Table VII and Table VIII. For each hierarchy height, we report the number of wins, loses, and draws for each Java system. Wins, loses, and draws represent the number of concerns[7] or methods[8] for which a variant of MULAB (with a given hierarchy height) outperforms the other variants, loses to another variant, and perform equally well as the other variants, respectively. We also report the overall results in the last row.

Table VII shows the data analysis results on effectiveness scores of MULAB with different hierarchy heights (L=1,2,3,4). From the table, we can see that the variant of MULAB with L set to 4 outperforms the others. Among the 136 concerns, MULAB with L=4 performs the best on 61 concerns, MULAB with L=3 performs the best on 35 concerns, MULAB with L=2 performs the best on 24 concerns, and MULAB with L=1 performs the best on 28 concerns. The effectiveness scores are the same for 16 concerns. So we can draw the conclusion that our default configuration (i.e., L=4) outperforms the other three. We also compare the worst performing variant (i.e., L=2) with PR, and the result shows that it outperforms PR for 88 concerns (draws for 42 concerns) in terms of effectiveness. Wilcoxon sign-rank test shows that the difference in the effectiveness scores is significant at p-value of < 0.001. The Cliff's d is 0.353 which corresponds to a medium effect size.

Table VIII shows the data analysis results on rank scores of MULAB with different hierarchy heights (L=1,2,3,4). From the table, we can see that the variant of MULAB with L set to 4 outperforms the others. Among the 388 methods, MULAB with L=4 performs the best on 137 methods, MULAB with L=3 performs the best on 89 methods, MULAB with L=2 performs the best on 99 methods, and MULAB with L=1 performs the best on 105 methods. The ranks scores are the same for 40 methods. So we can draw the conclusion that our default configuration (i.e., L=4) outperforms the other three. We also compare the worst performing variant (i.e., L=3) with PR, and the result shows that it outperforms PR for 259 methods (draws for 121 methods) in terms of rank. Wilcoxon sign-rank test shows that the difference in the rank scores is significant at p-value of < 0.001. The Cliff's d is 0.371 which corresponds to a medium effect size.

MULAB *with L=4 performs better than* MULAB *with L=1, L=2, and L=3. The worst performing variant among the four still statistically significantly and substantially outperforms* PR *in terms of effectiveness and rank.*

[7] When effectiveness is used as a yardstick

[8] When rank is used as a yardstick

TABLE VII
Data analysis results on effectiveness scores of MULAB with different hierarchy heights (L=1,2,3,4). #Wins = Number of concerns for which a variant of MULAB outperforms the others, #Loses = Number of concerns for which a variant MULAB loses from another variant, #Draws = Number of concerns for which all variants perform equally well.

	L=1			L=2			L=3			L=4		
System	**#Wins**	**#Loses**	**#Draws**	**#Wins**	**#Loses**	**#Draws**	**#Wins**	**#Loses**	**#Draws**	**#Wins**	**#Loses**	**#Draws**
Art of Illusion	2	6	0	0	8	0	4	4	0	2	6	0
aTunes	2	11	3	2	11	3	3	10	3	8	5	3
jEdit4.2	4	11	1	3	12	1	3	12	1	8	7	1
jEdit4.3	0	3	1	1	2	1	1	2	1	3	0	1
Cocoon	2	6	6	4	4	6	2	6	6	3	5	6
Derby	7	22	0	5	24	0	9	20	0	15	14	0
Lucene	5	14	5	3	16	5	7	12	5	10	9	5
OpenJPA	6	19	0	6	19	0	6	19	0	12	13	0
Overall	**28**	**92**	**16**	**24**	**96**	**16**	**35**	**85**	**16**	**61**	**59**	**16**

TABLE VIII
Data analysis results on rank scores of MULAB with different hierarchy heights (L=1,2,3,4). #Wins = Number of methods for which a variant of MULAB outperforms the others, #Loses = Number of methods for which a variant MULAB loses from another variant, #Draws = Number of methods for which all variants perform equally well.

	L=1			L=2			L=3			L=4		
System	**#Wins**	**#Loses**	**#Draws**	**#Wins**	**#Loses**	**#Draws**	**#Wins**	**#Loses**	**#Draws**	**#Wins**	**#Loses**	**#Draws**
Art of Illusion	2	10	0	2	10	0	4	8	0	4	8	0
aTunes	5	20	5	5	20	5	6	19	5	14	11	5
jEdit4.2	7	25	1	7	25	1	6	26	1	14	18	1
jEdit4.3	1	7	1	3	5	1	2	6	1	4	4	1
Cocoon	8	14	16	11	11	16	9	13	16	12	10	16
Derby	26	53	1	18	61	1	25	54	1	29	50	1
Lucene	33	64	15	31	66	15	26	71	15	33	64	15
OpenJPA	23	50	1	22	51	1	11	62	1	27	46	1
Overall	**105**	**243**	**40**	**99**	**249**	**40**	**89**	**259**	**40**	**137**	**211**	**40**

D. *Threats to Validity*

Threats to internal validity relate to errors in our experiments. We have double checked our implementations and all the experiment results. Hence, we believe there are minimal threats to internal validity. Still, there could be errors that we did not notice.

Threats to external validity relate to the generalizability of our results. We have tried to mitigate this threat by evaluating our approach on 136 concerns from 8 open source software systems. The software systems we used in our empirical study were chosen primarily because of the availability of data and previous studies. Data of these systems were manually vetted and a part of these systems were also used in previous work [11], [12], [31], [32]. Admittedly, the concerns that we investigate may not sufficiently represent all concerns from all systems. Finally, our choice of baseline clearly impacts the results. As future work, we plan to study more baselines.

Threats to construct validity refer to the suitability of our evaluation metrics. We use effectiveness and rank which are also used by past software engineering studies to evaluate the effectiveness of concern localization techniques [11], [12], [27], [28]. Thus, we believe there is little threat to construct validity.

V. Related Work

A. *Concern Localization*

Concern localization is an important and recurring step in maintenance of a software system. We describe some past studies in the following paragraphs. Due to space limitations, our survey is by no means complete.

Text analysis. Wang et al. [3] evaluate 10 information retrieval techniques and discover that VSM has the best performance. Rao and Kak also investigate the use of LDA with VSM [14]. However, in their approach, VSM is considered separately from LDA. The results of the two are combined together using a *weighted sum*. The performance of the resulting composite model is *worse* than that of VSM. In this work, we integrate LDA and VSM by constructing a single unified vector and we use a hierarchy of topic models; the resulting approach performs better than Scanniello et al.'s approach, which has been shown to be better than VSM on the same dataset [11].

Text and static analysis. To improve the accuracy of concern localization, a few hybrid approaches have been proposed, which combine IR techniques with static program analysis. Zhao et al. [33] present a two-phase approach to concern localization, which first applies an IR technique to identify an initial set of feature-code-unit links based on the textual description of the concerns and code units, and then enrich the initial links by exploring program call graph. Similarly, Eaddy et al. [34] employ pruned dependency analysis to boost the recall of IR or dynamic-analysis-based approaches. Most recently, Scanniello et al. [11] propose a text retrieval-based concern localization technique which considers the structural relationships between source code documents. They use a link analysis algorithm PageRank to rank the document space and to improve concern localization. The algorithm uses links (i.e., dependencies) among documents to organize them into a hierarchical structure. With their technique, source code documents are automatically ranked with respect to a textual query written by the developer, based on the dependencies and the lexical similarities between the documents. We have shown that our approach which relies only on textual contents of concerns and methods are able to outperform the latest approach by Scanniello et al. on a benchmark dataset used by many prior studies.

Text, static and/or dynamic analysis. Aside from text and information gleaned using static analysis, execution traces have been used to aid concern localization. Liu et al. [35] apply IR-based filtering to rank the methods being executed in a single test scenario. Dit et al. [36] define a data fusion model for feature location that integrates different types of information to locate features using IR, dynamic analysis, and web mining algorithms. Our technique does not consider execution traces since most bug reports and change requests do not come with execution traces [37].

B. Search-Based Algorithms in Software Engineering

Search-based algorithms have been used to improve various software engineering activities. Harman and Jones propose the concept of search-based software engineering and they demonstrate how to reformulate a SE problem as a search-based problem [38]. Later, Harman et al. provide a review and classification of search-based software engineering techniques [39]. Many search-based algorithms have been proposed in the literature; we highlight a number of them in the following paragraphs.

Li et al. use various search algorithms including greedy search, hill climbing, and genetic algorithms for test case prioritization [40]. Canfora et al. construct a classification model by using multi-objective genetic algorithm for cross-project defect prediction [41]. Wang et al. propose a search-based approach for clone detection [42]. A number of search-based 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 package [44]. Gold et al. reformulate concept binding problem (i.e., assigning the most plausible concept for a source code segment) as a search problem to allow overlapping concept boundaries, and genetic and hill climbing algorithms are used to search for solutions to this problem [45].

Mancoridis *et al.* use a search-based algorithm to group software modules into clusters by minimizing cohesion and maximizing coupling [46]. Wang et al. use a genetic algorithm to improve fault localization; their approach analyzes a set of failing and correct execution traces to locate faulty basic blocks that are root causes of bugs [47]. Goues et al. propose *GenProg*, which uses genetic algorithm to automatically repair defects in software projects [48]. Le et al. propose HDRepair that mines bug fix patterns from version history and subsequently uses genetic programming to evolve patches for new bugs based on mined fix patterns [49]. Le et al. propose to use program logic specifications to evolve a buggy implementation until a correct patch is found via genetic programming and deductive verification [50]. More recently, Panichella et al. use genetic algorithm to identify near optimal solutions to customize various stages of an IR process [51]. The proposed approach explores what kinds of character pruning, identifier splitting, stop word removal, stemming, term weighting, and IR techniques are best to be used. Lohar et al. present a novel approach to trace retrieval, which utilizes a machine-learning engine to search for the best configuration given an initial training set of validated trace links [52]. Wang et al. introduce desktop and parallelised cloud-deployed versions of a search-based solution that finds suitable configurations for empirical studies [42]. Xia et al. propose an accurate change classification technique named collective personalized change classification (CPCC), which leverages a multi-objective genetic algorithm [53]. They also utilize genetic algorithm to do cross-project defect prediction; in particular, they propose a hybrid model reconstruction approach, named HYDRA, which contains two phases: genetic algorithm (GA) phase and ensemble learning (EL) phase [54].

In this work, similar to the above approaches, we also utilize a search-based algorithm. However, we address a new problem, namely multi-abstraction concern localization. The approach by Panichella et al. [51] only considers one level of abstraction.

VI. Conclusion and Future Work

Existing concern localization studies characterize both concerns and code units as a bag of tokens at *one abstraction level*. In this study, we propose a multi-abstraction concern localization technique named MULAB which combines a hierarchy of topic models with VSM. We use genetic algorithm to estimate a near-optimal configuration of the topic models. Our experiments on 136 concerns from 8 open-source software systems show that our approach performs better than PR, the state-of-art approach recently proposed by Scanniello et al. [11], when evaluated in terms of effectiveness and rank. In the future, we plan to perform a deeper analysis on cases where our multi-abstraction approach does not work well, and improve the effectiveness of our proposed approach further. We also plan to merge our approach with other advanced text mining solutions, e.g., paraphrase detection, deep learning, etc., for more optimal performance.

Acknowledgment. This research was supported by NSFC Program (No.61572426) and National Key Technology R&D Program of the Ministry of Science and Technology of China under grant 2015BAH17F01.

References

[1] J. Anvik, L. Hiew, and G. C. Murphy, "Coping with an open bug repository," in *ETX*, 2005, pp. 35–39.

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

[3] S. Wang, D. Lo, Z. Xing, and L. Jiang, "Concern localization using information retrieval: An empirical study on linux kernel," in *WCRE 2011*.

[4] T.-D. B. Le, R. J. Oentaryo, and D. Lo, "Information retrieval and spectrum based bug localization: Better together," in *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*. ACM, 2015, pp. 579–590.

[5] S. Wang, D. Lo, and J. Lawall, "Compositional vector space models for improved bug localization," 2014.

[6] S. Wang and D. Lo, "Version history, similar report, and structure: Putting them together for improved bug localization," in *Proceedings of the 22nd International Conference on Program Comprehension*. ACM, 2014, pp. 53–63.

[7] X. Xia, D. Lo, X. Wang, C. Zhang, and X. Wang, "Cross-language bug localization," in *Proceedings of the 22nd International Conference on Program Comprehension*. ACM, 2014, pp. 275–278.

[8] 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 *2012 34th International Conference on Software Engineering (ICSE)*. IEEE, 2012, pp. 14–24.

[9] M. P. Robillard and G. C. Murphy, "Representing concerns in source code," *ACM Trans. Softw. Eng. Methodol.*, vol. 16, no. 1, 2007.

[10] D. M. Blei, A. Y. Ng, and M. I. Jordan, "Latent dirichlet allocation," *J. Mach. Learn. Res.*, vol. 3, pp. 993–1022, Mar. 2003.

[11] G. Scanniello, A. Marcus, and D. Pascale, "Link analysis algorithms for static concept location: an empirical assessment," *Empirical Software Engineering*, vol. 20, no. 6, pp. 1666–1720, 2015.

[12] G. Scanniello and A. Marcus, "Clustering support for static concept location in source code," in *Program Comprehension (ICPC), 2011 IEEE 19th International Conference on*. IEEE, 2011, pp. 1–10.

[13] 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 *Proceedings of the 2013 International Conference on Software Engineering*. IEEE Press, 2013, pp. 522–531.

[14] 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.

[15] T.-D. B. Duy, S. Wang, and D. Lo, "Multi-abstraction concern localization," in *Proceedings of the International Conference on Software Maintenance*. IEEE, 2013.

[16] D. M. Blei, A. Y. Ng, and M. I. Jordan, "Latent dirichlet allocation," *the Journal of machine Learning research*, vol. 3, pp. 993–1022, 2003.

[17] H. U. Asuncion, A. U. Asuncion, and R. N. Taylor, "Software traceability with topic modeling," in *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering-Volume 1*. ACM, 2010, pp. 95–104.

[18] S. W. Thomas, "Mining software repositories using topic models," in *Proceedings of the 33rd International Conference on Software Engineering*. ACM, 2011, pp. 1138–1139.

[19] T. L. Griffiths and M. Steyvers, "Finding scientific topics," *Proceedings of the National Academy of Sciences*, vol. 101, no. suppl 1, pp. 5228–5235, 2004.

[20] H. M. Wallach, D. M. Mimno, and A. McCallum, "Rethinking lda: Why priors matter," in *Advances in neural information processing systems*, 2009, pp. 1973–1981.

[21] J. H. Holland, *Adaptation in natural and artificial systems: an introductory analysis with applications to biology, control, and artificial intelligence.* U Michigan Press, 1975.

[22] D. E. Goldberg *et al.*, *Genetic algorithms in search optimization and machine learning*. Addison-wesley Reading Menlo Park, 1989, vol. 412.

[23] P. J. Rousseeuw and L. Kaufman, *Finding Groups in Data*. Wiley Online Library, 1990.

[24] J. Sander, M. Ester, H.-P. Kriegel, and X. Xu, "Density-based clustering in spatial databases: The algorithm gdbscan and its applications," *Data mining and knowledge discovery*, vol. 2, no. 2, pp. 169–194, 1998.

[25] A. Hotho, A. Maedche, and S. Staab, "Ontology-based text document clustering," *KI*, vol. 16, no. 4, pp. 48–54, 2002.

[26] C. Manning, P. Raghavan, and H. Schutze, *Introduction to Information Retrieval*. Cambridge, 2008.

[27] 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.

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

[29] F. Wilcoxon, "Individual comparisons by ranking methods," *Biometrics bulletin*, vol. 1, no. 6, pp. 80–83, 1945.

[30] N. Cliff, *Ordinal methods for behavioral data analysis*. Psychology Press, 2014.

[31] L. Moreno, W. Bandara, S. Haiduc, and A. Marcus, "On the relationship between the vocabulary of bug reports and source code," in *Proceedings of International Conference on Software Maintenance*, 2013, pp. 452–455.

[32] 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 International Conference on Software Engineering*, ser. ICSE. IEEE Press, 2013, pp. 842–851.

[33] W. Zhao, L. Zhang, Y. Liu, J. Sun, and F. Yang, "Sniafl: Towards a static noninteractive approach to feature location," *ACM Transactions on Software Engineering and Methodology (TOSEM)*, vol. 15, no. 2, pp. 195–226, 2006.

[34] M. Eaddy, A. V. Aho, G. Antoniol, and Y.-G. Guéhéneuc, "Cerberus: Tracing requirements to source code using information retrieval, dynamic analysis, and program analysis," in *Program Comprehension, 2008. ICPC 2008. The 16th IEEE International Conference on*. IEEE, 2008, pp. 53–62.

[35] D. Liu and S. Xu, "A combined concept location method for java programs," in *Computer Software and Applications Conference, 2007. COMPSAC 2007. 31st Annual International*, vol. 2. IEEE, 2007, pp. 29–42.

[36] B. Dit, M. Revelle, and D. Poshyvanyk, "Integrating information retrieval, execution and link analysis algorithms to improve feature location in software," *Empirical Software Engineering*, vol. 18, no. 2, pp. 277–309, 2013.

[37] C. Sun, D. Lo, S.-C. Khoo, and J. Jiang, "Towards more accurate retrieval of duplicate bug reports," in *ASE*, 2011, pp. 253–262.

[38] M. Harman and B. F. Jones, "Search-based software engineering," *Information and Software Technology*, vol. 43, no. 14, pp. 833–839, 2001.

[39] M. Harman, S. A. Mansouri, and Y. Zhang, "Search-based software engineering: Trends, techniques and applications," *ACM Computing Surveys (CSUR)*, vol. 45, no. 1, p. 11, 2012.

[40] Z. Li, M. Harman, and R. M. Hierons, "Search algorithms for regression test case prioritization," *IEEE Trans. Softw. Eng.*, vol. 33, no. 4, pp. 225–237, Apr. 2007. [Online]. Available: http://dx.doi.org/10.1109/TSE.2007.38

[41] G. Canfora, A. De Lucia, M. Di Penta, R. Oliveto, A. Panichella, and S. Panichella, "Multi-objective cross-project defect prediction," in *Software Testing, Verification and Validation (ICST), 2013 IEEE Sixth International Conference on*. IEEE, 2013, pp. 252–261.

[42] T. Wang, M. Harman, Y. Jia, and J. Krinke, "Searching for better configurations: a rigorous approach to clone evaluation," in *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering*. ACM, 2013, pp. 455–465.

[43] P. Tonella, "Evolutionary testing of classes," in *Proceedings of the 2004 ACM SIGSOFT international symposium on Software testing and analysis*, ser. ISSTA '04. New York, NY, USA: ACM, 2004, pp. 119–128. [Online]. Available: http://doi.acm.org/10.1145/1007512.1007528

[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 *Proceedings of the Software Metrics, 10th International Symposium*, ser. METRICS '04. Washington, DC, USA: IEEE Computer Society, 2004, pp. 172–183. [Online]. Available: http://dx.doi.org/10.1109/METRICS.2004.4

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

[46] S. Mancoridis, B. S. Mitchell, Y. Chen, and E. R. Gansner, "Bunch: A clustering tool for the recovery and maintenance of software system structures," in *Proceedings of the IEEE International Conference on Software Maintenance*, ser. ICSM '99. Washington, DC, USA: IEEE Computer Society, 1999, pp. 50–. [Online]. Available: http://dl.acm.org/citation.cfm?id=519621.853406

[47] S. Wang, D. Lo, L. Jiang, Lucia, and H. C. Lau, "Search-based fault localization." in *ASE*, P. Alexander, C. S. Pasareanu, and J. G. Hosking, Eds. IEEE, 2011, pp. 556–559.

[48] C. Le Goues, T. Nguyen, S. Forrest, and W. Weimer, "Genprog: A generic method for automatic software repair," *Software Engineering, IEEE Transactions on*, vol. 38, no. 1, pp. 54–72, 2012.

[49] X. D. Le, D. Lo, and C. Le Goues, "History driven program repair," in *IEEE 23rd International Conference on Software Analysis, Evolution, and Reengineering, SANER 2016, Suita, Osaka, Japan, March 14-18, 2016*, 2016, pp. 213–224.

[50] X. B. D. Le, Q. L. Le, D. Lo, and C. L. Goues, "Enhancing automated program repair with deductive verification," in *ICSME*, 2016.

[51] A. Panichella, B. Dit, R. Oliveto, M. D. Penta, D. Poshyvanyk, and A. D. Lucia, "Parameterizing and assembling ir-based solutions for software engineering tasks using genetic algorithms," in *SANER*, 2016.

[52] S. Lohar, S. Amornborvornwong, A. Zisman, and J. Cleland-Huang, "Improving trace accuracy through data-driven configuration and composition of tracing features," in *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering*. ACM, 2013, pp. 378–388.

[53] X. Xia, D. Lo, X. Wang, and X. Yang, "Collective personalized change classification with multiobjective search," *IEEE Transactions on Reliability*, 2016.

[54] X. Xia, D. Lo, S. J. Pan, N. Nagappan, and X. Wang, "Hydra: Massively compositional model for cross-project defect prediction," *IEEE Transactions on software Engineering*, 2016.