Improving Software Text Retrieval using Conceptual Knowledge in Source Code

Zeqi Lin, Yanzhen Zou, Junfeng Zhao, and Bing Xie
Key Laboratory of High Confidence Software Technologies, Ministry of Education, Beijing, China, 100871
School of Electronics Engineering and Computer Science, Peking University, Beijing, China, 100871
Beida(Binhai) Information Research, Tianjin, China, 300450
{linzq14,zouyz,zhaojf,xiebing}@sei.pku.edu.cn

***Abstract*—A large software project usually has lots of various textual learning resources about its API, such as tutorials, mailing lists, user forums, etc. Text retrieval technology allows developers to search these API learning resources for related documents using free-text queries, but it suffers from the lexical gap between search queries and documents. In this paper, we propose a novel approach for improving the retrieval of API learning resources through leveraging software-specific conceptual knowledge in software source code. The basic idea behind this approach is that the semantic relatedness between queries and documents could be measured according to software-specific concepts involved in them, and software source code contains a large amount of software-specific conceptual knowledge. In detail, firstly we extract an API graph from software source code and use it as software-specific conceptual knowledge. Then we discover API entities involved in queries and documents, and infer semantic document relatedness through analyzing structural relationships between these API entities. We evaluate our approach in three popular open source software projects. Comparing to the state-of-the-art text retrieval approaches, our approach lead to at least 13.77% improvement with respect to *mean average precision (MAP)*.**

***Index Terms*—Software text retrieval, conceptual knowledge, API graph, semantic relatedness.**

I. Introduction

Reusing APIs of existing software projects is a common practice during software development. A large software project usually has lots of various textual learning resources about its API, such as tutorials, mailing lists, user forums, etc. When developers have issues in reusing an API, they usually use text retrieval systems to search its API learning resources for documents that can help resolve these issues. For example, *StackOverflow* provides a text retrieval system for its users to search related questions.

Most existing software text retrieval systems are based on lexical similarity, especially the cosine similarity between TF-IDF vectors representing the query and the document in a vector space model[1]. This approach usually suffers from the lexical gap between queries and documents: a document related to a query may be long and contain many keywords which do not occur in the query. As a result, the document will be scored low and ranked poorly in the search results.

To address this problem, researchers pointed out that documents should be ranked according to not only lexical similarity, but also conceptual knowledge. Here *conceptual knowledge* is defined as concepts and entities in the real world and relationships between them. Recent years, with the rapid development of large-scale conceptual knowledge bases on the Internet (such as WordNet[2], DBpedia[3], Yago[4], etc), many researches ([5], [6], [7], [8], [9], [10], [11]) have proven that conceptual knowledge can improve text retrieval systems semantically. However, this idea is not easy to be applied in software text retrieval systems. It is because that: documents in API learning resources usually involve many software-specific concepts, while these software-specific concepts are rarely contained in universal conceptual knowledge bases on the Internet.

Our idea is that we can extract software-specific conceptual knowledge from software source code and leverage it to bridge the lexical gap between queries and documents. This is mainly because: (1) most software-specific concepts are defined as API entities (such as classes, interfaces and methods) in software source code; (2) the semantic relatedness among these software-specific concepts is reflected in structural dependencies (such as inheritances, declarations and method invocations) between API entities, and we can leverage it to infer the semantic relatedness between queries and documents.

In this paper, we present a novel approach for improving software text retrieval using software-specific conceptual knowledge in software source code. In detail, firstly we represent conceptual knowledge in source code as an API graph in which nodes represent API entities , and heterogeneous directed edges represent structural dependencies between them. Then, we discover API entities involved in each query and document using *Recodoc* [12]—a tool for recovering traceability links between an API and its learning resources—and several word matching based heuristic rules. Each query or document is represented as a weighted collection of API entities. After that, we leverage *TransR* [13], a recent multi-relational data embedding algorithm, to analyze the API graph so that the pairwise API entity similarity could be measured. For each document, we score its semantic relatedness to the

This paper is supported by National Key Research and Development Program of China (Grant No. 2016YFB1000801), National Natural Science Fund for Distinguished Young Scholars (Grant No. 61525201) and National Natural Science Foundation of China (Grant No. 61472007).

query based on the pairwise similarity of API entities involved in them. Finally, we use the score to re-rank search results in software text retrieval systems.

Comparing to the state-of-the-arts, our work makes the following contributions:

1) We represent each document in API learning resources as a weighted collection of API entities so that it can be "explained" by software-specific conceptual knowledge in source code.
2) We measure the semantic relatedness between queries and documents in terms of software-specific conceptual knowledge by leveraging multi-relational data embedding technology. We use it to improve document ranking in software text retrieval systems.
3) We evaluate our approach in three popular open source software projects. Comparing to the state-of-the-art approaches, our approach lead to at least 13.77% improvement with respect to *mean average precision (MAP)*.

The rest of this paper is organized as follows. We provide background on software text retrieval and conceptual knowledge-based text retrieval in Section 2. Section 3 presents an example to illustrate our motivation. Section 4 describes the details of our approach. Section 5 presents experiments that evaluate our approach. We conclude and discuss future work in Section 6.

II. Background and Related Work

We discuss background and related work from two aspects: the first is improving text retrieval in software engineering; the second is conceptual knowledge-based text retrieval approaches in the information retrieval community.

A. Software Text Retrieval

Text retreival is one of the most popular technologies used in software engineering, where it has been successfully applied to seek out textual information from various software artifacts, such as source code files (e.g., [14], [15], [16], [17], [18], [19]), software documentation (e.g., [20], [21], [22], [23]), bug reports (e.g., [24], [25]), questions in online question answering communities (e.g., [26], [27], [28], [29]), etc.

The performance of software text retrieval systems is usually suboptimal due to the lexical gap between queries and documents. To deal with this problem researchers have proposed many query reformulation approaches to improve software text retrieval. Haiduc et al. [30] proposed an automatic query reformulation approach called *Refoqus* for software text retrieval through learning from a sample of queries and relevant results. Panichella et al. [31] proposed an approach based on genetic algorithms for improving software text retrieval using topic models (e.g. Latent Semantic Indexing(LSI) and Latent Dirichlet Allocation (LDA)). Some other approaches have been proposed to build software-specific word similarity database and then use it to expand queries in software text retrieval. For example, Tian et al. [32], [33] proposed an approach for building software-specific word similarity database through mining StackOverflow questions and answers using Positive Pointwise Mutual Information (PPMI); Ye et al. [34] calculate semantic similarities between words from software documentation using word embedding techniques; Howard et al. [35] and Yang et al. [36] infer semantically related words from software source code context; Wang et al. [37] infer semantically related software terms and their taxonomy through mining tags in FreeCode; etc.

Besides these lexical features, a lot of rank assistant approaches have been proposed to improve document ranking in software text retrieval. For example, Ponzanelli et al. [26] proposed an approach for searching StackOverflow discussions. In this approach, question score, accepted answer score, user reputation and question tags are leveraged to rank search results. Ye et al. [19] proposed an approach for searching source code files relevant to bug reports. In this approach, features like bug-fixing recency and bug-fixing frequency are leveraged to rank search results. Zou et al. [38] proposed a question-oriented approach for improving document ranking in software text retrieval. In this approach, how search results are ranked depends on interrogatives (such as "*how*", "*why*" and "*which*") in the queries.

These approaches are effectiveness in their software text retrieval tasks. However, most of these approaches need large annotated datasets (usually are query-document pairs) for training. Comparing to them, our approach has the advantage that it does not need any annotated dataset.

B. Conceptual Knowledge-based Text Retrieval

Recent years, with the rapid development of large-scale universal machine-readable knowledge bases on the Internet (such as WordNet[1], DBpedia[2], Freebase[3], Google Knowledge Graph[4], Yago[5], etc), researchers have proposed many approaches to leverage conceptual knowledge in these knowledge bases to improve text retrieval (e.g., [5], [6], [7], [8], [9], [10], [11]). Here "*conceptual knowledge*" means concepts and entities in the real world and relationships between them, such as: <*Michael_Jackson, publish_song, Billi_Jean*>, <*Barack_Obama, born_in, Honolulu*>, etc. The basic process of leveraging conceptual knowledge to improve text retrieval is that: first, detect entities mentioned in documents; second, use relationships between these entities to measure semantic document relatedness; third, use the semantic document relatedness to improve document ranking in text retrieval. These researches prove that it is feasible to use conceptual knowledge to improve text retrieval. Inspired by these researches, we aim to improve software text retrieval using conceptual knowledge. However, in software text retrieval systems, documents are extracted from various software artifacts, thus lots of software-specific entities which are not contained in universal knowledge bases are mentioned in these documents. Therefore,

[1] http://wordnet.princeton.edu/wordnet/
[2] http://wiki.dbpedia.org/
[3] https://developers.google.com/freebase/
[4] https://developers.google.com/knowledge-graph/
[5] www.yago-knowledge.org/

TABLE I: TWO SAMPLE DOCUMENTS ON STACKOVERFLOW

Doc A: Lucene query parser to use filters for wildcard queries (The newly asked question)
Body: My problem is how to parse wildcard queries with Lucene that the query term is passed through a TokenFilter. I'm using a custom Analyzer with several filters (e.g. ASCIIFoldingFilter). My problem is that whether Lucene's QueryParser detects that one of the sub-queries is a WildcardQuery ...
Doc B: How to get a Token from a Lucene TokenStream? (A ground truth related question to Doc A)
Body: I'm trying to use Apache Lucene for tokenizing, and I am baffled at the process to obtain Tokens from a TokenStream. The worst part is that I'm looking at the comments in the JavaDocs of TokenStream.incrementToken() that address my question. Somehow, an AttributeSource is supposed to be used, rather than Tokens. I'm totally at a loss. Can anyone explain how to get token-like information from a TokenStream? ...

we need to look for software-specific conceptual knowledge resources.

Tang et al. [39] proposed an approach for improving document ranking in software text retrieval using conceptual knowledge. In this approach, the software-specific conceptual knowledge resources are domain ontologies constructed by domain experts. As most software projects do not have domain ontologies, the available of this approach is limited.

In software engineering community, structural information in software source code are successfully leveraged in many automatic software engineering tasks to help software developers learn and understand software projects during software development, maintenance and reuse (e.g. [40], [41], [42], [43], [44], [45], [46]). For example, McMillan et al. [41] studied how to leverage structural information in software source code as conceptual knowledge to recovery traceability links among requirement artifacts. In this paper, we further study how to leverage structural information in software source code as conceptual knowledge to improve the retrieval of API learning resources.

III. MOTIVATING EXAMPLE

In this section, we present two sample documents selected from *StackOverflow* to further motivate our research. These two sample documents are both questions about the API of *Apache Lucene*. The first question[6] (denoted as *Doc A*) is "*Lucene query parser to use filters for wildcard queries*", while the second question[7] (denoted as *Doc B*) is "*How to get a Token from a Lucene TokenStream?*". Table I shows titles and detailed descriptions of them. *Doc B* is a ground truth related question to *Doc A*, which was pointed out in the answer of *Doc A*.

Consider the scenario that *Doc A* is a newly asked question and the user want to find some questions related to it. In the keyword matching based text retrieval system provided by *StackOverflow*, *Doc B*, the ground truth related question to *Doc A*, is not ranked high, since it contains many keywords which do not occur in *Doc A*.

[6]http://stackoverflow.com/questions/28650774/lucene-query-parser-to-use-filters-for-wildcard-queries

[7]http://stackoverflow.com/questions/2638200/how-to-get-a-token-from-a-lucene-tokenstream

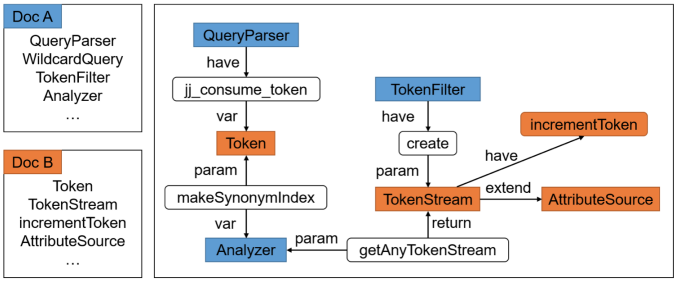


Fig. 1: A brief illustration of how we can use conceptual knowledge in source code to capture the semantic relatedness between two documents

To address this problem, we aim to leverage conceptual knowledge in source code to capture the semantic relatedness between *Doc A* and *Doc B*. This idea is mainly composed of two phases: first, we represent each document as a weighted collection of API entities so that it can be "explained" by software-specific conceptual knowledge in source code; then, we measure the semantic relatedness between these two documents in terms of software-specific conceptual knowledge.

For instance, consider this sentence in *Doc A*:

> *My problem is how to **parse wildcard queries** with Lucene that the **query term** is passed through a **TokenFilter**...*

Class *TokenFilter* is mentioned directly in this sentence. Moreover, as the document contains keywords *parse*, *wildcard*, *query* and *term*, it is reasonable to guess that class *WildcardQuery*, class *QueryParser* and class *Term* are involved in this sentence as well. Based on this idea, we present *Doc A* as a collection of API entities. Similarly, we present *Doc B* as a collection of API entities containing class *Token*, class *TokenStream*, method *TokenStream.incrementToken()*, class *AttributeSource*, etc. We assign weights to these API entities based on term frequency-inverse document frequency (TF-IDF).

Fig. 1 illustrates how we can use conceptual knowledge in software source code to capture the semantic relatedness between *Doc A* and *Doc B*. This figure shows a partial API graph of Apache Lucene, in which API entities involved in *Doc A* are colored blue, and API entities involved in *Doc B* are colored orange. Though *Doc B* contains many keywords which do not occur in *Doc A*, we can capture the semantic relatedness between them for API entities involved in them are structurally related in the API graph.

To sum up, there are two major technical issues to be addressed: 1) how a document should be represented as a weighted collections of API entities; 2) how the relatedness between two weighted collections of API entities should be scored.

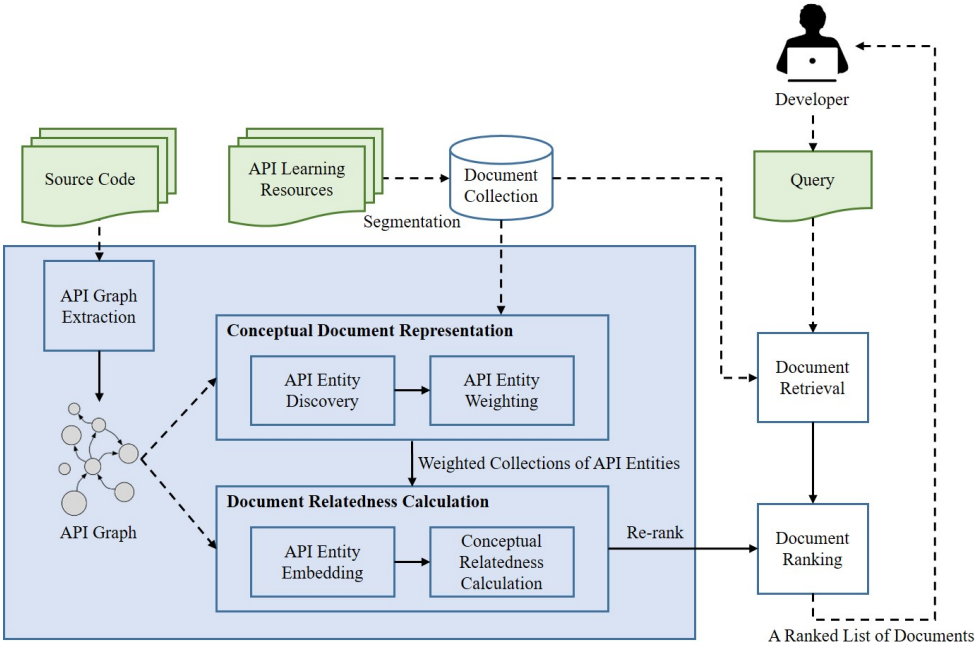


Fig. 2: Workflow of our approach for improving software text retrieval using conceptual knowledge in source code

IV. Approach

In this paper, we present a novel re-ranking approach for improving the retrieval of related API learning resources (or "software text retrieval" for short) using software-specific conceptual knowledge in software source code. Fig. 2 shows the workflow of our approach. It consists of three phases: (1) API graph extraction, (2) conceptual document representation, and (3) document relatedness calculation.

- *API Graph Extraction.* This phase aims to extract an API graph from source code of the software project. An API graph consists of API entities (such as classes, interfaces and methods) in the software source code and various structural dependencies (such as inheritances, declarations and method invocations) between them. In our approach, the API graph is used as the software-specific conceptual knowledge to capture the semantic relatedness between queries and documents.
- *Conceptual Document Representation.* This phase aims to represent each query or document as a weighted collection of API entities. In detail, firstly we discover API entities involved in the document (or query) using *Recodoc* [12] approach and several keyword matching based heuristic rules. Then, we assign weights to these API entities based on TF-IDF.
- *Document Relatedness Calculation.* This phase measures the semantic relatedness between queries and documents based on the conceptual document representations. Firstly, we leverage *TransR* [13], a recent multi-relational data embedding algorithm, to analyze the API graph so that the pairwise API entity similarity is measured. Then, we use the pairwise API entity similarity as natural building blocks to calculate the conceptual relatedness between two weighted collections of API entities. We use the result to score how much a document is semantically related to the query. In software text retrieval systems, we re-rank the search results based on the following rules: if a document gains a higher score, it will be re-ranked towards the top, and conversely it will be ranked lower down.

A. API Graph Extraction

In this paper, we use *ASTParser* in *org.eclipse.jdt.core*[8] to extract an API graph from source code of a Java project. In an API graph, nodes represent API entities in the software source code, and heterogeneous directed edges represent structural dependencies between them. We define four different types of API entities: *class*, *interface*, *method* and *field*, and we define eleven different types of structural dependencies between them (see Table II).

B. Conceptual Document Representation

This phase aims to represent each query or document as a weighted collection of API entities. Here "document" means not only text segments in API learning resources, but also queries. This phase is composed of two parts: API entity discovery and API entity weighting.

1) API Entity Discovery: The most important thing in the conceptual document representation phase is to discover API entities involved in documents. Firstly, we discover API entities directly mentioned in documents using *Recodoc* [12]. Then, as API entities directly mentioned in documents are far away from enough for conceptual document representation,

[8]https://mvnrepository.com/artifact/org.eclipse.jdt/org.eclipse.jdt.core

TABLE II: ELEVEN DIFFERENT EDGE TYPES IN API GRAPH

Edge Type	Description
extend	Inheritance of a child class/interface to its parent
implement	A class to interfaces it implements
haveMethod	A class/interface to its member methods
haveField	A class to its member fields
throw	A method to types of exceptions it throws
fieldType	A field to its type
parameterType	A method to types of its parameters
variableType	A method to types of variable objects constructed in it
returnType	A method to types of its return objects
methodInvocation	A method to methods invoked in it
fieldInvocation	A method to fields invoked in it

we discover more API entities involved in documents based on keyword matching.

Recodoc is a tool for recovering traceability links between an API and its learning resources. We use it to discover API entities directly mentioned in documents. For example, consider the sentence in the motivating example: "*My problem is how to parse wildcard queries with Lucene that the query term is passed through a TokenFilter...*". *Recodoc* will extract class *TokenFilter* from this sentence.

However, API entities discovered by *Recodoc* are far away from enough for conceptual document representation. In the above example sentence, *Recodoc* discovers only one API entity: class *TokenFilter*. It is unacceptable for us to represent the sentence using only class *TokenFilter*, since the majority of information (such as keywords *parse*, *wildcard*, *query*, *term*, etc) in the sentence is neglected. Therefore, we need to discover more API entities for the sentence through leveraging these keywords. For example, it is reasonable to guess that class *WildcardQuery* should be added to the collection of API entities for the sentence, since this sentence contains keywords *wildcard* and *query*. Similarly, class *QueryParser* and class *Term* should be discovered as well. Though these API entities are not directly mentioned in the sentence, they represent the conceptual information hidden in its keywords, which is important for the conceptual document representation. Therefore, we propose a keyword matching based method for discovering API entities in documents besides those discovered by *Recodoc*.

In this method, a document is firstly split into words using non-word characters (such as white characters and punctuations). Then, we filter English stopwords (such as "*a*", "*in*", "*us*", etc) out, and stem the rest of words using *Porter Stemming Algorithm*. After these lexical pre-processings, we get the keyword set of the document.

For each keyword, we define its candidate API entities as API entities whose identifiers contain the keyword. The number of candidate API entities for a keyword may be large. For example, there are 55 candidate API entities for keyword *english*, such as class *EnglishStemmer*, class *EnglishAnalyzer*, class *EnglishPossessiveFilterFactory*, etc. Taking such a large number of API entities into the consideration of conceptual document representation will not only increase the time cost, but also hinder the performance of subsequent processes. Therefore, we propose several heuristic rules based on our observations to reduce the number of candidate API entities, including:

- Filter some API entities out according to keywords in their identifiers. For example, "*filter*" is a keyword in a document, and class *TokenFilter* and class *StopFilter* are two of its candidate API entities. The document contains another keyword "*token*" as well, but it does not contain keyword "*stop*". the percentage of matched keywords in *TokenFilter* is higher than that in *StopFilter*. Therefore, we filter class *StopFilter* out and keep class *TokenFilter*.
- Filter some API entities out according to their types. We regard that concepts in the software project are mainly defined in classes/interfaces. Therefore, if the candidate API entity collection contains both classes/interfaces and methods/fields, we will filter methods and fields out from it.
- Filter some API entities out according to the lengths of their identifiers. For example, class *EnglishAnalyzer* and class *EnglishMinimalStemFilterFactory* are two candidate API entities for keyword "english". We filter class *EnglishMinimalStemFilterFactory* because its identifier is much longer than class *EnglishAnalyzer*. In our approach, we define "much longer" as five letters longer or twice longer.

For each keyword in the document, we obtain a collection of remaining candidate API entities. We define the final collection of API entities for the document as the union of these collections.

2) API Entity Weighting: We assign weights to keywords in the document using TF-IDF. After that, we re-assign the weights of keywords to API entities corresponding to them:

$$
\begin{aligned}
w_c &= \frac{w'_c}{\sum\limits_{c' \in C} w'_{c'}} \\
w'_c &= \sum_{t \in T_c} \frac{tfidf_t}{|C_t|}
\end{aligned}
\tag{1}
$$

In this equation, w_c represents the weight assigned to API entity c; C represents all API entities discovered in the document; T_c represents all keywords correspond to API entity c; C_t represents all API entities correspond to keyword t.

After API entity discovery and API entity weighting, each document is represented as a weighted collection of API entities. We use the weighted collection as the conceptual representation of the document.

C. Document Relatedness Calculation

This phase is used to score the semantic relatedness between a query and a document using their conceptual document representations so as to re-rank search results in software text retrieval systems. It is composed of two parts: API entity embedding and conceptual relatedness calculation.

1) API Entity Embedding: This step aims to measure pairwise similarity between different API entities in the API graph. It is an off-line step.

In different software engineering tasks, the pairwise similarity between API entities has different definitions. For example, in code clone detection, the pairwise similarity between two API entities means how code fragments in them are similar to each other; in concept location, the pairwise similarity between two API entities means how they are structurally dependent to each other; etc. In our approach, we define the pairwise similarity between two API entities as how their contexts in the API graph are similar to each other. In this definition, "context" means how the API entity is linked to other API entities in the API graph. For example, if class *A* and class *B* are both sub-classes of class *C*, the pairwise similarity between class *A* and class *B* will increase; if these two classes are both used as parameter types of method *m* together, the pairwise similarity between them will increase as well. Moreover, we regard that the pairwise similarity between API entities should be transitive in the API graph. For example, suppose that the pairwise similarity between class *A* and class *B* is high, and method m_1 and method m_2 uses them respectively. In this case, we think method the pairwise similarity between method m_1 and method m_2 should gain some increase.

Given two API entities, our approach calculates how their contexts in the API graph are similar to each other based on multi-relational data embedding technology. Multi-relational data embedding is a technology for learning vector representations of entities from multi-relational data (i.e., data consisting of entities and different types of relationships between them) [47]. First, each entity is represented as a random vector in a shared, low-dimensional, continuous vector space. Then, gradient descent methods are used to optimize these vectors: if two entities have the same kind of relationships to the same entity, their vectors will be "drawn closer" to each other; otherwise their vectors will "pull away" from each other. After the global optimization, these vectors of entities are finally adjusted to proper positions in the shared vector space, and we can measure the pairwise similarity between two entities based on the distance between their vectors. In our approach, the API graph is a form of multi-relational data, thus we leverage multi-relational data embedding methods to learn vector representations of API entities from it. More specifically, we use *TransR* [13], a widely used multi-relational data embedding method, for API entity embedding. We normalize the distance between two API entities to the range of $[0, 1]$ and denote it as $dist(e_1, e_2)$. Then we define the pairwise similarity between these two API entities $sim(e_1, e_2)$ as $(1 - dist(e_1, e_2))$.

2) Conceptual Relatedness Calculation: To compute the conceptual relatedness between two weighted collections of API entities C_q (i.e., the query) and C_d (i.e., the document), we modify the text-to-text similarity measure introduced by Mihalcea et al [48]. The conceptual relatedness between an API entity c and a weighted collection of API entities C is defined as the maximum similarity between c and any API entity c' in C:

$$sim(c, C) = \max_{c' \in C} sim(c, c') \tag{2}$$

An asymmetric similarity $sim(C_q \rightarrow C_d)$ is defined as weighted sum of similarities between API entities in C_q and the entire collection of API entities in C_D:

$$sim(C_q \rightarrow C_d) = \sum_{c \in C_q} sim(c, C_d) * w_c \tag{3}$$

The asymmetric similarity $sim(C_d \rightarrow C_q)$ is defined analogously, by swapping C_Q and C_D in the formula above. Finally, the symmetric similarity $sim(C_q, C_d)$ betweem two weighted collections of API entities C_q and C_d is defined as the sum of the two asymmetric similarities:

$$sim(C_q, C_d) = sim(C_q \rightarrow C_d) + sim(C_d \rightarrow C_q) \tag{4}$$

We use $sim(C_q, C_d)$ to represent the conceptual relatedness between query Q and document D, and then use it to re-score search results in software text retrieval systems. The new document scoring function is defined as:

$$S(d) = \alpha \cdot S_0(d) + (1 - \alpha) \cdot sim(C_q, C_d) \tag{5}$$

In this document scoring function, $S_0(d)$ represents the original document scoring function in the software text retrieval system. Usually, $S_0(d)$ is defined as the lexical similarity between the query and the document, for example, the cosine similarity between TF-IDF vectors representing the query and the document in a vector space model. We normalize $S_0(d)$ to the range of $[0, 1]$, and then combine it with $sim(C_q, C_d)$ linearly.

V. Empirical Evaluation

To evaluate our approach, we have conducted a set of quantitative experiments. These experiments address the following questions:

RQ1. Does our approach improve the retrieval of API learning resources?

The objective of our approach is to improve the retrieval of API learning resources. We are concerned about how well our approach works. To evaluate it, firstly we study the effect of varying hyper-parameters in our approach. After that, we make some comparisons between our approach and existing text retrieval approaches.

RQ2. How does the conceptual document representation component work in our approach?

In our approach, we represent each document as a weighted collection of API entities. We are concerned about the effectiveness of these conceptual document representations. Therefore, we conduct experiments to study how different conceptual document modeling strategies will impact the re-ranking results.

RQ3. How does the semantic relatedness calculation component work in our approach?

TABLE III: VARIOUS STATISTICS FROM TEST DATA

Project Name	Source Code Version	API Entities	Relationships	Documents	Keywords/Doc	Entities/Doc
Apache Lucene	6.3.0	19,701	65,216	4,753	64.13	85.96(7.48)
Apache POI	3.14	37,594	87,629	1,348	70.85	90.10(18.36)
JFreeChart	1.0.19	12,655	32,974	1,078	78.98	38.89(21.60)

In our approach, we embed API entities in the API graph as low-dimensional continuous vectors, then we calculate pairwise similarity between two different API entities. We are concerned about the effectiveness of these pairwise similarities. Therefore, we conduct experiments to study how different methods for measuring pairwise similarity between two different API entities will impact the re-ranking results.

A. Evaluation Setting Up

We evaluate our approach in a real-world software text retrieval task for three popular open source Java projects: *Apache Lucene*[9], *Apache POI*[10] and *JFreeChart*[11], respectively.

1) Document Collections: *StackOverflow* provides good API learning resources for different software projects, thus we build the document collections for the evaluation using *StackOverflow* data dump. For each of the three projects, we download all questions and answers about it from StackOverflow according to their tags. Among these data, we model each question with its accepted answer as a document. Therefore, we build a document collection for each of the three software projects respectively. Table III shows the number of documents each software project contains (the *Documents* column) and the average number of keywords the the document collection of each software project contains (the *Keywords/Doc* column).

2) API Graph Preparation: For each project, our approach extracts an API graph from its source code. The project have many versions of source code, while we select the newest version among them for API graph extraction. Table III shows the source code version number (the *Source Code Version* column), the number of extracted API entities(the *API Entities* column) and the number of extracted relationships among these API entities(the *Relationships* column).

In our approach, we represent each document as a weighted collection of API entities. The *Entities/Doc* column in Table III shows the average size of these collections. For example, "85.96(7.48)" means that: on average, the conceptual representation of each document contains 85.96 API entities, and 7.48 of them are directly mentioned in the document.

3) Test Queries: To evaluate our approach, we conduct text retrieval process with 300 real-world queries from *StackOverflow* questions. Each project has 100 test queries. Since our goal is to help software developers reuse APIs, we restrict that these 100 questions should be about the project's API. It means that we rule out questions like "How to install this software?" or "Should I use this software or that software" when selecting these 100 questions. For each of these questions, we split it into words, stem them, filter stop words out, and finally get keywords in it. These keywords are regarded as a test query for searching the document collection.

[9]http://lucene.apache.org/
[10]http://poi.apache.org/
[11]http://www.jfree.org/jfreechart/

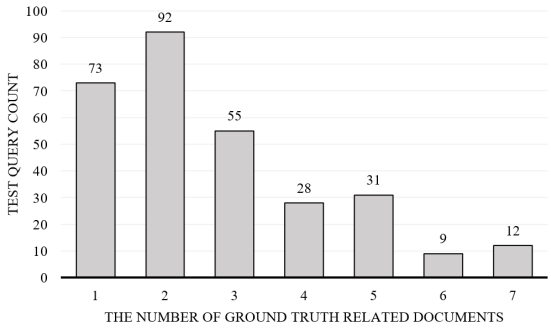


Fig. 3: The distribution of the number of ground truth related documents

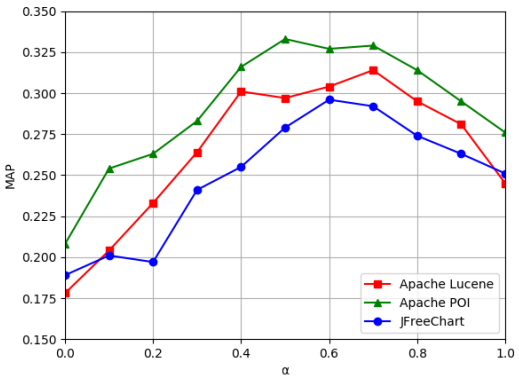


Fig. 4: The effect of varying α on the MAP of our approach

4) Ground Truth Related Documents: We build a standard text retrieval system based on the cosine similarity between TF-IDF vectors representing the query and the document in a vector space model. For each test query, we use the text retrieval system to obtain the top-20 documents related to it (documents corresponding to these test queries are excluded from the document collections). The evaluation task is to re-rank these documents. We ask three undergraduate students majoring in computer science to annotate these query-document pairs. They are all familiar with Java programming and these three software projects. For each test query, we show the corresponding question and its accepted answer to annotators. Then, each annotator judges whether each of the 20 retrieved documents is helpful to answer the question (mainly according to the comparison of the accepted answer and the retrieved document). If a query-document pair is annotated

"YES" by all the three annotators, we regard the document as a ground truth related document to the query. Fig. 3 shows the distribution of the number of ground truth related documents. On average, each test query has 2.76 ground truth documents related to it.

5) Baselines: We denote our approach as *CK*, i.e., *C*onceptual *K*nowledge and compare it against four text retrieval methods:

- *TF-IDF*. This is the baseline method. It ranks documents based on the cosine similarity between TF-IDF vectors representing the query and the document in a vector space model [1].
- *Okapi BM25*. Okapi BM25 is a widely used ranking function that extends TF-IDF in a probabilistic way [49]. We use the Okapi BM25 function provided by *Apache Lucene* to compute Okapi BM25 scores.
- *Latent Dirichlet Allocation (LDA)*. LDA is a popular generative model for text documents that learns representations for documents as distributions over word topics [50]. We use the LDA library provided by *mallet*[12] to train LDA models. Then, we score a query-document pair using the cosine similarity of their latent topic distributions.
- *Word Embedding (word2vec)*. This method learns word embeddings from document corpora and uses the distance between word embeddings as natural building blocks to score a query-document pair [34].

Many supervised learning based approaches have been proposed for improving software text retrieval, such as [28], [29], [30] and [38]. We do not compare our approach against these approaches because they need large annotated datasets for training. In practice, it is usually difficult to collect and annotate enough query-document pairs for a given software project.

6) Evaluation Metric: We use the *Mean Average Precision (MAP)*[51] and the *Mean Reciprocal Rank (MRR)*[52] as our evaluation metrics. MAP and MRR are both standard evaluation metrics in text retrieval.

Mean average precision for a set of queries is the mean of the average precision scores for each query, as shown in Equation 6 and 7.

$$MAP = \frac{\sum_{q=1}^{Q} AveP(q)}{Q} \tag{6}$$

$$AveP(q) = \frac{\sum_{k=1}^{n} (P(k) \times rel(k))}{number\ of\ related\ documents} \tag{7}$$

where k is the rank in the sequence of retrieved documents, $P(k)$ is the precision at cut-off k in the list, and $rel(k)$ is an indicator function equaling 1 if the item at rank k is a related document, zero otherwise. For example, consider that a query has two ground truth documents related to it. They are ranked at the 2nd and the 5th places respectively. The average precision score (aveP) of this query is: $\frac{1/2+2/5}{2} = 0.45$.

Mean reciprocal rank for a set of queries is the harmonic mean of ranks of the first related documents, as shown in Equation 8.

$$MRR = \frac{\sum_{q=1}^{Q} \frac{1}{first(q)}}{Q} \tag{8}$$

B. RQ1: The Comparison with Other Approaches

In our approach, we combines the conceptual document relatedness with the original document scoring function linearly to rank search results. α is a hyper-parameter that represents the weight of the original document scoring function. For $\alpha = 0$, our approach uses only the conceptual document relatedness; for $\alpha = 1$, our approach amounts to the standard TF-IDF based text retrieval approach. Fig. 4 presents the effect of varing α on the MAP of our approach. Our first observation is that our approach outperforms the standard TF-IDF based text retrieval approach significantly. Take the dataset of *Apache Lucene* as an example. We get the highest MAP value (0.314) When we set $\alpha = 0.7$, while the MAP value of the standard TF-IDF based text retrieval approach ($\alpha = 1$) is 0.245. Furthermore, Fig. 4 shows that it is necessary to combine the conceptual document relatedness with the original lexical document similarity. If we re-rank retrieved documents using only the conceptual document relatedness (i.e., $\alpha = 0$), the MAP results will be substantially lower down. For the three software projects, the best performed α settings are 0.7, 0.5, 0.6, respectively.

In our approach, there is another hyper-parameter: the vector dimension d [13] in the API entity embedding step. We vary d from 100 to 500, and the experimental results show that this hyper-parameter has limited effect on the performance of our approach. Therefore, we set $d = 200$ in the evaluation.

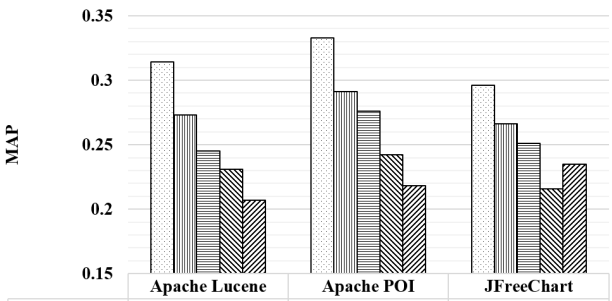
Fig. 5 compares our approach (CK) with four other text retrieval approaches: TF-IDF, Okapi BM25, LDA and Word2vec. Table IV shows the average MAP and MRR results of these approaches across all the three software projects, as well as average MAP and MRR improvement results relative to TF-IDF. In Fig. 5 and Table IV, all these approaches are ranked according to their performance. Comparing to the standard text retrieval system (TF-IDF), our approach obtains an improvement of 22.2% in MAP, while the second best performed approach, word2vec, obtains an improvement of 7.4% in MAP. Therefore, for RQ1., we can conclude that our approach outperforms the state-of-the-arts.

C. RQ2: Conceptual Document Representation Evaluation

We compare the conceptual document representation component in our approach with the following two alternatives:

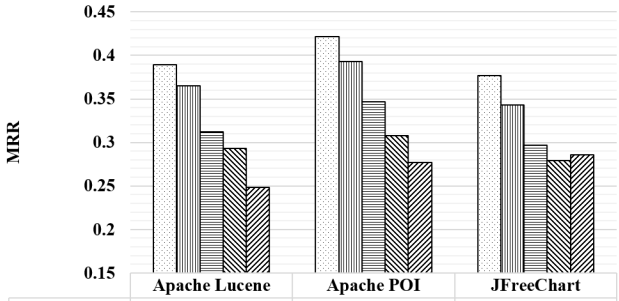
- *CK-RecodocOnly*. This approach is the same as *CK*, except that in this approach the weighted collection of API entities for each document contains only API entities

[12]http://mallet.cs.umass.edu/



	Apache Lucene	Apache POI	JFreeChart
CK	0.314	0.333	0.296
Word2vec	0.273	0.291	0.266
TF-IDF	0.245	0.276	0.251
LDA	0.231	0.242	0.216
Okapi BM25	0.207	0.218	0.235

(a) MAP



	Apache Lucene	Apache POI	JFreeChart
CK	0.389	0.422	0.377
Word2vec	0.365	0.393	0.343
TF-IDF	0.312	0.347	0.297
LDA	0.293	0.308	0.279
Okapi BM25	0.248	0.277	0.286

(b) MRR

Fig. 5: Average MAP and MRR comparison results

TABLE IV: AVERAGE MAP AND MRR COMPARISON RESULTS ACROSS ALL THE THREE SOFTWARE PROJECTS

	CK	word2vec	TF-IDF	Okapi BM25	LDA
MAP	**0.314**	0.276	0.257	0.230	0.220
MAP Improvement (w.r.t. TF-IDF)	**22.2%**	7.4%	0	-10.5%	-14.4%
MRR	**0.396**	0.367	0.319	0.293	0.270
MRR Improvement (w.r.t. TF-IDF)	**24.1%**	15.0%	0	-8.15%	-15.4%

directly mentioned in the document discovered by *Recodoc*. We compare our approach with *CK-RecodocOnly* to study the effectiveness of the keyword matching based API entity expansion step in our approach.

- *CK-UnWeighted*. This approach is the same as *CK*, except that in this approach API entities discovered for each document are not weighted. We compare our approach with *CK-UnWeighted* to study the effectiveness of the TF-IDF based API entity weighting step in our approach.

Fig. 6 shows the comparison results. Our approach outperforms *CK-RecodocOnly* and *CK-UnWeighted* significantly in all the three software projects. For example, in *Apache Lucene*, our approach gets the highest MAP value 0.314 when $\alpha = 0.7$, while the highest MAP value is 0.257 for *CK-RecodocOnly* ($\alpha = 0.9$) and 0.279 for *CK-UnWeighted* ($\alpha = 0.8$). These experimental results prove that:

- API entities directly mentioned in documents discovered by *Recodoc* are not enough to represent concepts involved in these documents. It is necessary to expand the collection of API entities through keyword matching.
- API entities involved in documents should be weighted based on TF-IDF so that the conceptual document representations can be more effective.

D. RQ3: Semantic Document Relatedness Evalution

We compare the semantic document relatedness component in our approach with the following two alternatives:

- *CK-Coupling*. This approach is the same as *CK*, except that in this approach we calculate the pairwise similarity between two different API entities using the coupling measurement suit proposed by Briand et al. [53].
- *CK-ShortestPath*. This approach is the same as *CK*, except that in this approach we calculate the pairwise similarity between two different API entities based on the *Dijkstra* shortest path distance between them in the API graph [7].

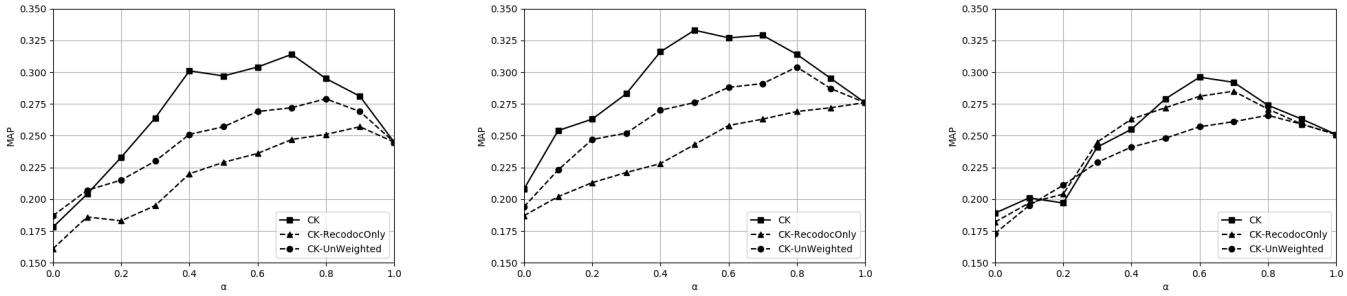
Fig. 7 shows the comparison results. Our approach outperforms *CK-Coupling* and *CK-ShortestPath* significantly in all the three software projects. These two alternatives perform poorly. For example, in *Apache Lucene*, *CK-ShortestPath* gets the highest MAP value 0.248 when $\alpha = 0.8$, which is just slightly better than the standard TF-IDF based text retrieval approach. For *Coupling*, the best performed α setting is 1, which means that the re-ranked search results are even worse than the original search results. These experimental results prove that: to capture the semantic document relatedness, the pairwise API entity similarity should be defined as how their contexts in the API graph are similar to each other (*CK*), rather than how they are structurally dependent to each other (*CK-Coupling* and *CK-ShortestPath*).

E. Threats to Validity

There are a number of threats to the validity of our results.

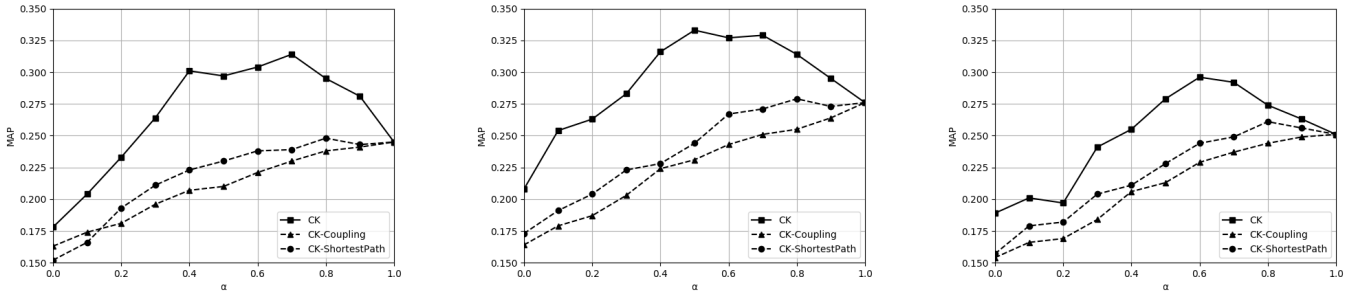
1) Construct Validity: Are the evaluation metrics suitable? We evaluate our approach using MAP and MRR, which are both standard evaluation metrics in text retrieval. We do not use precision at some cutoff (Precision@k), since the number of ground truth related documents of a test query may be less than k.

2) Internal Validity: Are there any experimental biases? First, the test query construction and ground truth related document annotation. We extract test queries from real-world



(a) Apache Lucene (b) Apache POI (c) JFreeChart

Fig. 6: The comparison between different conceptual document representation strategies



(a) Apache Lucene (b) Apache POI (c) JFreeChart

Fig. 7: The comparison between different API entity pairwise similarity calculation strategies

questions about software APIs in *StackOverflow*, rather than use designed test queries. We ask three undergraduate students majoring in computer science to annotate ground truth documents related to test queries. A document is regarded as a ground truth related document to a test query only if all the three annotators admit it. Second, the parameter settings in our approach. We conduct detailed experiments in which the main hyper-parameter α varies from 0 to 1, with the step size of 0.1. The bias is minimized by these efforts.

3) External Validity: Could the results be generalized? First, we conduct experiments in three popular open source software projects respectively to ensure that the results can be generalized in different software projects. Second, we construct the evaluation dataset using real-world API learning resources extracted from *StackOverflow*. For each project, we evaluate our approach using 100 test queries. The sizes of the document collections for the three projects are 4,753, 1,348, 1,078, respectively. These queries and documents are written by many different people, and they are all free-text without special formatting rules. Therefore, the results can be generalized in different document writing styles.

VI. Conclusion and Future Work

In this paper, we present a novel approach for improving software text retrieval using software-specific conceptual knowledge in software source code. We extract an API graph from software source code and use it as software-specific conceptual knowledge of the software project. Then, we represent each query or document as a weighted collection of API entities in the API graph. After that, we calculate the semantic relatedness between queries and documents through leverage multi-relational data embedding technology. Finally, we use the semantic relatedness to re-rank search results in software text retrieval systems. Detailed experiments are conducted on StackOverflow dataset to evaluate the effectiveness of our approach. Comparing to the state-of-the-art text retrieval approaches, our approach leads to at least 13.77% improvement with respect to *mean average precision (MAP)*.

In the future, we plan to further study how to improve conceptual document representation and API entity embedding in the proposed approach. In parallel, we will explore how this approach could be adapted to more types of textual artifacts in software, such as bug reports, source code comments, requirement specifications, etc.

References

[1] Chowdhury, Gobinda. Introduction to modern information retrieval. Facet publishing, 2010.

[2] Miller, George A. "WordNet: a lexical database for English." Communications of the ACM 38, no. 11 (1995): 39-41.

[3] Lehmann, Jens, Robert Isele, Max Jakob, Anja Jentzsch, Dimitris Kontokostas, Pablo N. Mendes, Sebastian Hellmann et al. "DBpediaa large-scale, multilingual knowledge base extracted from Wikipedia." Semantic Web 6, no. 2 (2015): 167-195.

[4] Hoffart, Johannes, Fabian M. Suchanek, Klaus Berberich, and Gerhard Weikum. "YAGO2: A spatially and temporally enhanced knowledge base from Wikipedia." Artificial Intelligence 194 (2013): 28-61.

[5] Gabrilovich, Evgeniy, and Shaul Markovitch. "Computing semantic relatedness using wikipedia-based explicit semantic analysis." In IJcAI, vol. 7, pp. 1606-1611. 2007.

[6] Egozi, Ofer, Shaul Markovitch, and Evgeniy Gabrilovich. "Concept-based information retrieval using explicit semantic analysis." ACM Transactions on Information Systems (TOIS) 29, no. 2 (2011): 8.

[7] Schuhmacher, Michael, and Simone Paolo Ponzetto. "Knowledge-based graph document modeling." In Proceedings of the 7th ACM international conference on Web search and data mining, pp. 543-552. ACM, 2014.

[8] Liu, Xitong, and Hui Fang. "Latent entity space: a novel retrieval approach for entity-bearing queries." Information Retrieval Journal 18, no. 6 (2015): 473-503.

[9] Xiong, Chenyan, and Jamie Callan. "Esdrank: Connecting query and documents through external semi-structured data." In Proceedings of the 24th ACM International on Conference on Information and Knowledge Management, pp. 951-960. ACM, 2015.

[10] Raviv, Hadas, Oren Kurland, and David Carmel. "Document retrieval using entity-based language models." In Proceedings of the 39th International ACM SIGIR conference on Research and Development in Information Retrieval, pp. 65-74. ACM, 2016.

[11] Ni, Yuan, Qiong Kai Xu, Feng Cao, Yosi Mass, Dafna Sheinwald, Hui Jia Zhu, and Shao Sheng Cao. "Semantic documents relatedness using concept graph representation." In Proceedings of the Ninth ACM International Conference on Web Search and Data Mining, pp. 635-644. ACM, 2016.

[12] Dagenais, Barthlmy, and Martin P. Robillard. "Recovering traceability links between an API and its learning resources." In Software Engineering (ICSE), 2012 34th International Conference on, pp. 47-57. IEEE, 2012.

[13] Lin, Yankai, Zhiyuan Liu, Maosong Sun, Yang Liu, and Xuan Zhu. "Learning Entity and Relation Embeddings for Knowledge Graph Completion." In AAAI, pp. 2181-2187. 2015.

[14] Marcus, Andrian, Andrey Sergeyev, Vaclav Rajlich, and Jonathan I. Maletic. "An information retrieval approach to concept location in source code." In Reverse Engineering, 2004. Proceedings. 11th Working Conference on, pp. 214-223. IEEE, 2004.

[15] Poshyvanyk, Denys, Yann-Gael Gueheneuc, Andrian Marcus, Giuliano Antoniol, and Vaclav Rajlich. "Feature location using probabilistic ranking of methods based on execution scenarios and information retrieval." IEEE Transactions on Software Engineering 33, no. 6 (2007).

[16] Zhou, Jian, Hongyu Zhang, and David Lo. "Where should the bugs be fixed?-more accurate information retrieval-based bug localization based on bug reports." In Proceedings of the 34th International Conference on Software Engineering, pp. 14-24. IEEE Press, 2012.

[17] Dit, Bogdan, Meghan Revelle, and Denys Poshyvanyk. "Integrating information retrieval, execution and link analysis algorithms to improve feature location in software." Empirical Software Engineering 18, no. 2 (2013): 277-309.

[18] Saha, Ripon K., Matthew Lease, Sarfraz Khurshid, and Dewayne E. Perry. "Improving bug localization using structured information retrieval." In Automated Software Engineering (ASE), 2013 IEEE/ACM 28th International Conference on, pp. 345-355. IEEE, 2013.

[19] Ye, Xin, Razvan Bunescu, and Chang Liu. "Learning to rank relevant files for bug reports using domain knowledge." In Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering, pp. 689-699. ACM, 2014.

[20] Lucia, De. "Information retrieval models for recovering traceability links between code and documentation." In Software Maintenance, 2000. Proceedings. International Conference on, pp. 40-49. IEEE, 2000.

[21] Antoniol, Giuliano, Gerardo Canfora, Gerardo Casazza, Andrea De Lucia, and Ettore Merlo. "Recovering traceability links between code and documentation." IEEE transactions on software engineering 28, no. 10 (2002): 970-983.

[22] Marcus, Andrian, and Jonathan I. Maletic. "Recovering documentation-to-source-code traceability links using latent semantic indexing." In Software Engineering, 2003. Proceedings. 25th International Conference on, pp. 125-135. IEEE, 2003.

[23] Oliveto, Rocco, Malcom Gethers, Denys Poshyvanyk, and Andrea De Lucia. "On the equivalence of information retrieval methods for automated traceability link recovery." In Program Comprehension (ICPC), 2010 IEEE 18th International Conference on, pp. 68-71. IEEE, 2010.

[24] Wang, Xiaoyin, Lu Zhang, Tao Xie, John Anvik, and Jiasu Sun. "An approach to detecting duplicate bug reports using natural language and execution information." In Software Engineering, 2008. ICSE'08. ACM/IEEE 30th International Conference on, pp. 461-470. IEEE, 2008.

[25] Nguyen, Anh Tuan, Tung Thanh Nguyen, Tien N. Nguyen, David Lo, and Chengnian Sun. "Duplicate bug report detection with a combination of information retrieval and topic modeling." In Proceedings of the 27th IEEE/ACM International Conference on Automated Software Engineering, pp. 70-79. ACM, 2012.

[26] Ponzanelli, Luca, Gabriele Bavota, Massimiliano Di Penta, Rocco Oliveto, and Michele Lanza. "Mining StackOverflow to turn the IDE into a self-confident programming prompter." In Proceedings of the 11th Working Conference on Mining Software Repositories, pp. 102-111. ACM, 2014.

[27] Ghaderi, Rezvan. "Improving the Retrieval of Related Questions in StackOverflow." (2015).

[28] Xu, Bowen, Deheng Ye, Zhenchang Xing, Xin Xia, Guibin Chen, and Shanping Li. "Predicting semantically linkable knowledge in developer online forums via convolutional neural network." In Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering, pp. 51-62. ACM, 2016.

[29] Bogdanova, Dasha, Ccero Nogueira dos Santos, Luciano Barbosa, and Bianca Zadrozny. "Detecting Semantically Equivalent Questions in Online User Forums." In CoNLL, vol. 123, p. 2015. 2015.

[30] Haiduc, Sonia, Gabriele Bavota, Andrian Marcus, Rocco Oliveto, Andrea De Lucia, and Tim Menzies. "Automatic query reformulations for text retrieval in software engineering." In Software Engineering (ICSE), 2013 35th International Conference on, pp. 842-851. IEEE, 2013.

[31] Panichella, Annibale, Bogdan Dit, Rocco Oliveto, Massimiliano Di Penta, Denys Poshyvanyk, and Andrea 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, pp. 522-531. IEEE Press, 2013.

[32] Tian, Yuan, David Lo, and Julia Lawall. "Automated construction of a software-specific word similarity database." In Software Maintenance, Reengineering and Reverse Engineering (CSMR-WCRE), 2014 Software Evolution Week-IEEE Conference on, pp. 44-53. IEEE, 2014.

[33] Tian, Yuan, David Lo, and Julia Lawall. "SEWordSim: Software-specific word similarity database." In Companion Proceedings of the 36th International Conference on Software Engineering, pp. 568-571. ACM, 2014.

[34] Ye, Xin, Hui Shen, Xiao Ma, Razvan Bunescu, and Chang Liu. "From word embeddings to document similarities for improved information retrieval in software engineering." In Proceedings of the 38th International Conference on Software Engineering, pp. 404-415. ACM, 2016.

[35] Howard, Matthew J., Samir Gupta, Lori Pollock, and K. Vijay-Shanker. "Automatically mining software-based, semantically-similar words from comment-code mappings." In Proceedings of the 10th Working Conference on Mining Software Repositories, pp. 377-386. IEEE Press, 2013.

[36] Yang, Jinqiu, and Lin Tan. "Inferring semantically related words from software context." In Proceedings of the 9th IEEE Working Conference on Mining Software Repositories, pp. 161-170. IEEE Press, 2012.

[37] Wang, Shaowei, David Lo, and Lingxiao Jiang. "Inferring semantically related software terms and their taxonomy by leveraging collaborative tagging." In Software Maintenance (ICSM), 2012 28th IEEE International Conference on, pp. 604-607. IEEE, 2012.

[38] Zou, Yanzhen, Ting Ye, Yangyang Lu, John Mylopoulos, and Lu Zhang. "Learning to rank for question-oriented software text retrieval (t)." In Automated Software Engineering (ASE), 2015 30th IEEE/ACM International Conference on, pp. 1-11. IEEE, 2015.

[39] Tang, Wenhu, Long Yan, Zhen Yang, and Qinghua Henry Wu. "Improved document ranking in ontology-based document search engine using evidential reasoning." IET software 8, no. 1 (2014): 33-41.

[40] Warr, Frederic Weigand, and Martin P. Robillard. "Suade: Topology-based searches for software investigation." In Proceedings of the 29th international conference on Software Engineering, pp. 780-783. IEEE Computer Society, 2007.

[41] McMillan, Collin, Denys Poshyvanyk, and Meghan Revelle. "Combining textual and structural analysis of software artifacts for traceability link recovery." In Traceability in Emerging Forms of Software Engineering, 2009. TEFSE'09. ICSE Workshop on, pp. 41-48. IEEE, 2009.

[42] Scanniello, Giuseppe, and Andrian Marcus. "Clustering support for static concept location in source code." In Program Comprehension (ICPC), 2011 IEEE 19th International Conference on, pp. 1-10. IEEE, 2011.

[43] Chan, Wing-Kwan, Hong Cheng, and David Lo. "Searching connected api subgraph via text phrases." In Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering, p. 10. ACM, 2012.

[44] Mcmillan, Collin, Denys Poshyvanyk, Mark Grechanik, Qing Xie, and Chen Fu. "Portfolio: Searching for relevant functions and their usages in millions of lines of code." ACM Transactions on Software Engineering and Methodology (TOSEM) 22, no. 4 (2013): 37.

[45] Panichella, Annibale, Collin McMillan, Evan Moritz, Davide Palmieri, Rocco Oliveto, Denys Poshyvanyk, and Andrea De Lucia. "When and how using structural information to improve ir-based traceability recovery." In Software Maintenance and Reengineering (CSMR), 2013 17th European Conference on, pp. 199-208. IEEE, 2013.

[46] Scanniello, Giuseppe, Andrian Marcus, and Daniele Pascale. "Link analysis algorithms for static concept location: an empirical assessment." Empirical Software Engineering 20, no. 6 (2015): 1666-1720.

[47] Bordes, Antoine, Nicolas Usunier, Alberto Garcia-Duran, Jason Weston, and Oksana Yakhnenko. "Translating embeddings for modeling multi-relational data." In Advances in neural information processing systems, pp. 2787-2795. 2013.

[48] Mihalcea, Rada, Courtney Corley, and Carlo Strapparava. "Corpus-based and knowledge-based measures of text semantic similarity." In AAAI, vol. 6, pp. 775-780. 2006.

[49] Robertson, Stephen E., Steve Walker, Susan Jones, Micheline M. Hancock-Beaulieu, and Mike Gatford. "Okapi at TREC-3." Nist Special Publication Sp 109 (1995): 109.

[50] Blei, David M., Andrew Y. Ng, and Michael I. Jordan. "Latent dirichlet allocation." Journal of machine Learning research 3, no. Jan (2003): 993-1022.

[51] Manning, Christopher D., Prabhakar Raghavan, and Hinrich Schtze. Introduction to information retrieval. Vol. 1, no. 1. Cambridge: Cambridge university press, 2008.

[52] Voorhees, Ellen M. "The TREC-8 Question Answering Track Report." In Trec, vol. 99, pp. 77-82. 1999.

[53] Briand, Lionel, Prem Devanbu, and Walcelio Melo. "An investigation into coupling measures for C++." In Proceedings of the 19th international conference on Software engineering, pp. 412-421. ACM, 1997.