Expanding Queries for Code Search Using Semantically Related API Class-names

Feng Zhang, Haoran Niu, Iman Keivanloo, *Member, IEEE*, and Ying Zou, *Member, IEEE*

Abstract—When encountering unfamiliar programming tasks (*e.g.*, connecting to a database), there is a need to seek potential working code examples. Instead of using code search engines, software developers usually post related programming questions on online Q&A forums (*e.g.*, Stack Overflow). One possible reason is that existing code search engines would return effective code examples only if a query contains identifiers (*e.g.*, class or method names). In other words, existing code search engines do not handle natural-language queries well (*e.g.*, a description of a programming task). However, developers may not know the appropriate identifiers at the time of the search. As the demand of searching code examples is increasing, it is of significant interest to enhance code search engines. We conjecture that expanding natural-language queries with their semantically related identifiers has a great potential to enhance code search engines. In this paper, we propose an automated approach to find identifiers (in particular API class-names) that are semantically related to a given natural-language query. We evaluate the effectiveness of our approach using 74 queries on a corpus of 23,677,216 code snippets that are extracted from 24,666 open source Java projects. The results show that our approach can effectively recommend semantically related API class-names to expand the original natural-language queries. For instance, our approach successfully retrieves relevant code examples in the top 10 retrieved results for 76% of 74 queries, while it is 36% when using the original natural-language query; and the median rank of the first relevant code example is increased from 22 to 7.

Index Terms—Query Expansion, Code Search, Neural Network Language Model, API Class-name

1 Introduction

During the process of software development, developers need to accomplish different programming tasks, such as connecting to a database, and uploading a file to a server. Many approaches (*e.g.*, [6, 16, 35]) have been proposed to help developers search relevant source code. For examples, GitHub and Koders are two code search engines that return source code snippets for given keywords. The keywords are usually identifiers, such as class and method names. Recent work by Stolee et al. [33] describes a new search model other than the keyword based engines, *i.e.*, let developers specify the behaviour of the target code, such as defining the expected input and output.

The programming tasks encountered by one developer might have already been encountered by others [11]. Developers often seek help from other developers for potential solutions (*e.g.*, code examples). A code example is a snippet of source code that has a few lines of code and is usually used to show how a programming task is implemented [19]. To get a potential solution from a large community of developers, developers often ask programming questions on online Q&A forums (*e.g.*, Stack Overflow, hereby SO) [11]. Such programming questions are presented in the form of natural language. For instance, when a developer looks for code examples that show the implementation of playing sound, he or she may ask the following question: *"How can I play sound using Clip in Java?"* (see Fig. 1). The increasing popularity of online Q&A forums indicates the increasing need of searching code examples. However, existing code search engines mainly support keyword-based queries that consist of a set of search terms (*e.g.*, class or method names). Therefore, we think there is a significant interest to enhance existing code search engines by expanding natural-language queries with identifiers (*e.g.*, class and method names). In addition, an improved code search engine can help companies to better manage code reuse and share knowledge in large scale proprietary software systems (*e.g.*, a car typically has about 100 million lines of code [27]).

Our preliminary study (see Section 2) shows that the proportion of identifiers plays a key role in retrieving relevant code examples from code search engines. As an example shown in Fig. 1, after adding five identifiers (*i.e.*, *"Player"*, *"AudioInputStream"*, *"LineUnavailableException"*, *"Game"*, and *"AudioClip"*) to the initial query, a code example that better matches the initial query is returned. There are various approaches to find terms (*e.g.*, identifiers) for query expansion in the software engineering area, and they can be grouped into two categories [14]: 1) semi-automated approaches that extract terms (*e.g.*, identifiers) only from the answers that have been marked by devel-

- *F. Zhang is with the School of Computing, Queen's University, Kingston, Canada. E-mail: feng@cs.queensu.ca.*
- *H. Niu, I. Keivanloo and Y. Zou are with the Department of Electrical and Computer Engineering, Queen's University, Kingston, Canada. E-mail: {13hn, ying.zou}@queensu.ca, and iman.keivanloo@ieee.org.*

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TSE.2017.2750682, IEEE Transactions on Software Engineering

Initial query:
How can I play sound using Clip in Java?

⇓

The first code example returned for the initial query:

```
1  private static boolean playSound(String soundUrl)
2          throws SlickException
3  {
4      boolean resourceFound = true;
5      Sound sound;
6      try {
7          sound = new Sound(soundUrl);
8      } catch (SlickException e) {
9          System.out.println("Did not find the
10                 following resource:"+soundUrl);
11         sound = new Sound("sound/error.ogg");
12         resourceFound = false;
13     }
14     sound.play();
15     while (sound.playing()) {
16         try { Thread.sleep(100);
17         } catch (InterruptedException e) {
18             e.printStackTrace();
19         }
20     }
21     return resourceFound;
22 }
```

Expanded query:
How can I play sound using Clip in Java? + Player AudioInputStream LineUnavailableException Game AudioClip

⇓

The first code example returned for the expanded query:

```
1  public static void playAudioClip(String clipFilename) {
2      Clip m_clip;
3      AudioInputStream audioInputStream = null;
4      try {
5          audioInputStream = AudioSystem.getAudioInputStream
6                      (new FileInputStream(clipFilename));
7          if (audioInputStream != null) {
8              AudioFormat format =
9                      audioInputStream.getFormat();
10             DataLine.Info info =
11                     new DataLine.Info(Clip.class, format);
12             m_clip = (Clip) AudioSystem.getLine(info);
13             m_clip.open(audioInputStream);
14             m_clip.start();
15         } else {
16             System.out.println("ClipPlayer.<init>(): can't
17                     get data from file "+clipFilename);
18         }
19     } catch (LineUnavailableException lineException) {
20         Toolkit.getDefaultToolkit().beep();
21     } catch (Exception e) { e.printStackTrace(); }
22 }
```

Fig. 1. The first code example returned for the initial (left) and the expanded (right) queries, respectively.

opers as relevant to the initial query; and 2) automated approaches that use natural language processing (NLP) models trained from the general English text to identify terms that are semantically related to the initial query [43].

In this paper, we propose an automated approach that recommends semantically related API[1] class-names to expand a natural-language query through a neural network language model. We focus on API class-names because applications with similar functionalities often use the same set of API classes. The approach is based on a continuous bag-of-words model [25] that can learn the semantic representations of API class-names from the source code. We compute the semantic distances between the terms of the initial query and API class-names, and select the top semantically related API class-names to expand the initial query.

Our major contributions are presented as follows:

- Conduct a preliminary study using a representative sample of 384 SO questions. We observe that code search related questions are prevalent in SO, and such questions are described in natural language with a small proportion of identifiers (*e.g.*, class names).
- Propose an automated approach to expand natural-language code search queries with semantically related API class-names, using a neural network language model.
- Evaluate our approach using a corpus of 23,677,216 code snippets. The results show that our approach can significantly enhance code search engines on natural-language queries. Specifically, the median position of the first correct answer in the returned list is lifted from 22 to 2 for the 74 studied queries. The improvement is desirable since developers always only look at the top results.

Paper organization. Section 2 shows our motivating studies. Section 3 describes the details of our approach. The case study design, results, and the threats to validity of our study are presented in Sections 4, 5, and 6, respectively. We discuss related work Section 7, and conclude in Section 8.

2 Motivating Study

In this section, we perform a preliminary study to find if it is desirable to support natural-language queries in code search engines and what is a possible way to expand natural-language queries.

RQ1. Is it common to seek code examples by posting descriptive questions on StackOverflow?

Motivation. As code search engines do not support natural-language queries well, developers often seek help from fellow peers by posting descriptive questions on online Q&A forums (*e.g.*, SO). In this research question, we classify questions posted on SO into various categories and quantify the number of questions related to code search. The prevalence of code search questions could indicate that it is a common practice for developers to search for code examples with only descriptive questions on SO, and therefore

1. API (application program interface) is a set of common classes and methods that are generally used to create software applications.

it is desirable to support natural-language queries in code search engines.

Approach. To address this preliminary question, we intended to restrict the questions related to "Java", which is the most popular object-oriented programming language [37]. We extract 725,492 questions from Stack Overflow that were tagged with "Java" and were posted between August 2008 to October 2014. To reflect the distribution of the 725,492 SO questions at the 95%±5% confidence level, a sample set of 384 questions are needed [3, 7]. Therefore, we randomly select 384 questions from the entire population to create a representative sample.

We manually review and classify the sampled questions. Inspired by the types of questions that are identified by Stolee et al. [33], we categorize questions into the following six categories:

(C1) *Code search* that contains questions on how to fulfil a programming task.
(C2) *Concept explanation* that includes questions regarding some programming concepts or techniques.
(C3) *Debugging* that has questions dealing with the run-time errors or unexpected exceptions.
(C4) *Procedure* that contains questions seeking a scenario to accomplish a certain task which is not related to programming.
(C5) *Suggestion* that includes questions looking for the availability of certain tools.
(C6) *Discussion* that has questions discussing performance or programming languages, etc.

Finally, we compute the proportion of the questions belonging to each of the six categories. The proportion can reflect the popularity of questions related to *Code search* on Stack Overflow.

Findings. The dominant category of questions on SO is *Code search*. The distribution of questions in each category is shown in Table 1. We observe that 31% of the sample questions belong to the category *Code search*. This observation indicates that SO is frequently used by developers when seeking solutions for a particular programming task. However, questions posted on forums like SO are answered by peer developers. It is not guaranteed that each question can be correctly answered in a timely manner. On the other hand, code search engines instantly return answers for a query. If using *Code search* questions as queries is well supported by code search engines, it could considerably relieve programming burdens of developers.

We observe that *Code search* related questions describe a particular programming task in a natural-language way. For instance, to complete the task of subtracting three months from a given object of class "java.util.Date", an example question is asked as: "java.util.Date - Deleting three months from a date?"[2]. There is no consistent pattern of expressing the *Code search* questions. For example, there are only 29% (*i.e.*, 34 out of 118) of *Code search* questions contain the word "how" in the sampled 384 questions. In the aforementioned example question, there is an identifier "java.util.Date". However, we observe that identifiers do not necessarily appear in questions, such as "Converting alphanumeric to ascii and incrementing"[3].

TABLE 1
The number of sampled questions of each category.

Category of questions	Number of questions (%)
(C1) Code search	118 (31%)
(C2) Concept explanation	26 (7%)
(C3) Debugging	62 (16%)
(C4) Procedure	104 (27%)
(C5) Suggestion	39 (10%)
(C6) Discussion	35 (9%)

Developers tend to prefer answers with code examples. We find that the accepted answers of *Code search* questions on SO usually contain code examples. Particularly, 62% (73 out of 118) of *Code search* questions contain code examples. This ratio is close to the observation by Stolee et al. [33], which is 63%.

To accomplish a particular programming task, it is a common practice of developers to seek code examples on SO by asking descriptive questions. Supporting to use such questions as queries in code search engines potentially accelerates the process of solution finding.

RQ2. Does including identifiers help code search?

Motivation. By exploring Stack Overflow, we observe that *Code search* questions are usually composed of natural-language terms, and do not necessarily contain identifiers (*i.e.*, class or method names). When using *Code search* questions as queries, our intuitive solution to improve the search efficiency is to extend the initial question with appropriate identifiers. In this question, we aim to examine if our intuitive solution makes sense, *i.e.*, if including identifiers in queries can help find code examples.

Approach. To address this question, we analyze a large number of code search queries with and without identifiers. The queries are extracted from a year-long (*i.e.*, from 2007-01-01 to 2007-12-31) usage log data of Koders.com[4], which was a very popular code search engine with over three million unique visitors in 2007 [1]. The usage log data was collected by

2. http://stackoverflow.com/questions/1311143/java-util-date-deleting-three-months-from-a-date
3. http://stackoverflow.com/questions/1383132/converting-alphanumeric-to-ascii-and-incrementing
4. Koders (now becomes "Open Hub") : http://code.openhub.net

Bajracharya and Lopes [1], and there are 628,862 code search queries in the data.

The usage log records two types of activities of users: 1) a search activity represents a user's query; and 2) a download activity happens if a user clicks one of the query results to view the code snippet. A download activity indicates that the user is interested on the clicked code snippet and uses the code snippet in some way.

We consider that a query is answered if at least one code snippet is clicked. The underlying assumption is that a user would click the code snippet only when the user thinks a code snippet is relevant to his/her query. This assumption has been used in prior studies (*e.g.*, [1, 17, 18]) on both code search and general purpose search for finding relevant results.

To determine if a query contains identifiers, we examine if a query contains any of the three fields: "mdef", "cdef" and "idef" that are used by Koders.com to specify method, class and interface names in a query, respectively.

To evaluate whether including identifiers in code search queries impacts the efficiency of code search engines, we split the 628,862 Koders queries into two groups: 1) one group contains all queries having one or more identifiers; and 2) the other group contains all remaining queries. Then we test the following null hypothesis, using the Chi-squared test [13] with the 95% confidence level (*i.e.*, p-value < 0.05):

H_{20}: the proportion of answered queries does not differ between the two groups of queries with and without identifiers.

The Chi-squared test assesses if there is a relationship between two categorical variables [13]. If there is a significant difference, we reject the null hypothesis and further compute odds ratio (OR) [30]. The odds ratio measures how strongly the presence of identifiers in queries associated with the success of getting answers. $OR = 1$ indicates that the presence of identifiers does not affect the search efficiency; $OR > 1$ shows that the presence of identifiers can increase the likelihood of having a query answered; and $OR < 1$ means that the presence of identifiers negatively affects the search efficiency.

Findings. Including identifiers (*i.e.*, class or method names) in a query significantly increases the chance for retrieving relevant code examples. The p-value of the Chi-squared test is less than $2.2e$-16, thus we reject the null hypothesis H_{20} and conclude that the proportion of answered queries is significantly different between the groups with or without identifiers. Table 2 presents the detailed number of queries.

The odds ratio is 2.23 (*i.e.*, $OR > 1$). This indicates that using queries with identifiers is 2.23 times more likely to get relevant code examples from code search engines than using queries without identifiers. The ratio of successful (*i.e.*, answered) to unsuccessful (*i.e.*, unanswered) queries is 32% among queries with identifiers, which is much higher than the one among queries without identifiers (*i.e.*, 14.5%). This observation motivates us to expand *Code search* questions with identifiers, towards helping developers find relevant code examples with natural-language queries.

TABLE 2
The number of answered and unanswered queries in groups of queries with or without identifiers.

	Answered (%)	**Unanswered (%)**	$\frac{Answered}{Unanswered}$
Queries with identifiers	5,090 (0.8%)	15,810 (2.5%)	32%
Queries without identifiers	76,847 (12.2%)	531,115 (84.5%)	14.5%

Including identifiers can significantly improve the search efficiency.

Summary of the motivating study. When performing a particular programming task, a common way for developers to find relevant code examples is to post *Code search* questions on SO. However, questions on SO are not guaranteed to be answered immediately. To get instant answers (*i.e.*, relevant code examples), it is desirable to use *Code search* questions as queries on code search engines.

Moreover, we observe that having identifiers in a query significantly improves the search efficiency. Considering that *Code search* questions are usually described in a natural-language way, extending the questions with identifiers may improve the efficiency of searching code examples with *Code search* questions. Therefore, we propose an approach to automatically expand natural-language queries with the semantically related identifiers (particularly API class-names).

3 Our Approach

In this section, we present the details of our approach for query expansion.

3.1 Overview

Fig. 2 depicts the overview of our approach. Our approach contains three major steps:

1) extracting vector representations that are used to map natural-language queries with identifiers;
2) applying the model to find relevant identifiers to expand the given query;
3) executing the expanded query to retrieve code snippet lists from the code snippet corpus.

3.2 Extracting Vector Representations

To map natural-language queries with identifiers, we extract vector representations of words by applying the continuous bag-of-words model (CBOW) [25].

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TSE.2017.2750682, IEEE Transactions on Software Engineering

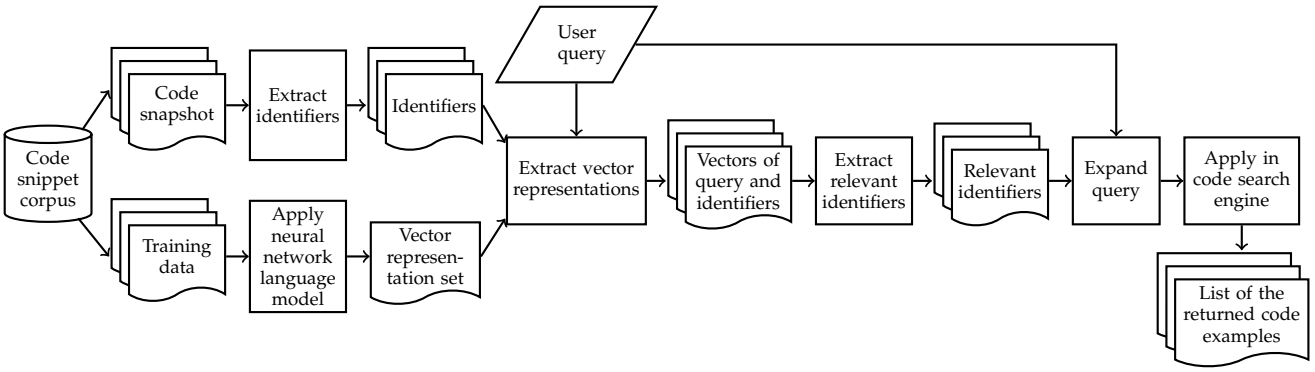

Fig. 2. The process of expanding natural-language queries.

Specifically, the CBOW model is used to find if two words describe the similar functionality of code.

The CBOW model is a neural network model that can infer semantically similar words from the context of words (*i.e.*, their surrounding words). For instance, we assume that the corpus contains ("the", "girl", "was", "walking") and ("the", "boy", "was", "walking"). The CBOW model learns that "girl" and "boy" are semantically similar, as they have the same context ("the", "was", "walking"). More background on the neural natural-language model is presented in Appendix **??**.

The details of extracting vector representations are described as follows.

1) Extracting identifiers. Identifiers include class and method names. Comparing to user-defined classes or API methods, API classes are more likely to appear in many code examples. In the other words, using API classes can retrieve more relevant code examples. Therefore, we choose to use only the API class-names as identifiers to expand the original natural-language queries. Specifically, we only use the classes that are described in Java API documentation.

To extract API class-names from a Java file, we parse all "import" statements to get the class-name. The "import" statement (*e.g.*, "import java.util.Arrays") specifies the Java class that is used in a Java file, and comprises of both the package name and the class name. The "import" statements that do not specify the Java class, such as "import java.util.*", are excluded. We process all Java files to extract all API class-names that exist in our corpus. In total, 15,085 class-names are extracted, including 97% (*i.e.*, 3,892 out of 4,024) of the Java API classes listed in the Java API doc.

2) Creating the training data for the CBOW model. We tokenize each code snippet (*i.e.*, a complete method) in the corpus through camel case splitting. The camel case splitting (*e.g.*, "DescriptionAttribute" to "Description" and "Attribute") has been used in prior studies [20, 31] to extract words from identifiers. As we propose to extend natural-language queries with API class-names, we keep them as is. Therefore, a term is either a word or an intact class-name.

We normalize the tokenized terms by removing stop words and stemming. Our stop words include both common English stop-words (*e.g.*, "a", "the", and "is") and programming keywords (*e.g.*, "if", "else", and "for"). We use the Porter stemmer for stemming the tokenized terms, such as converting the words "stems", "stemmer", "stemming" and "stemmed" to the word "stem". All the normalized terms make up the corpus for training the CBOW model.

3) Extracting vector representations. We choose to use Word2vec[5] that is an efficient implementation of the CBOW model [25]. Word2vec takes the corpus of the normalized terms as the training data to build the model, and produces a set of term vectors as the output.

The resulting term vectors include the vectors of all the terms that appear in the training data. Each element of a term vector is a float value, which reflects one semantic feature of the term.

3.3 Expanding the Natural-Language Queries

To expand queries, we first need to obtain the vector representation of a natural-language query. Then based on similarities between the query and API class-names, we find relevant API class-names from the corpus. Details are described as follows.

1) Obtaining the vector representation of a natural-language query. For each natural-language query, we perform the camel case splitting, stop word removal and stemming. We use a vector $V_q = (t_1, t_2, ..., t_i, ..., t_m)$ to represent a normalized natural-language query, where m is the total number of terms in the normalized query, and t_i represents the i_{th} term. For example, the natural-language query "How can I play sound using Clip in Java?" can be represented by the vector V_q = ("plai", "sound", "us", "clip").

The terms contained in the natural-language queries are likely to appear in the training data, which is created using a large-scale code snippet corpus. Then for each term t_i in the normalized query, we extract its term vector representation (*i.e.*, a float

5. word2vec: http://code.google.com/p/word2vec/

vector) from term vectors that are generated on the training data. We denote the term vector as $t_i = (f_{i1}, f_{i2}, ..., f_{ij}, ..., f_{in})$, where n is the total number of semantic features, and f_{ij} is the j_{th} semantic feature.

We obtain the vector representation T_q of the normalized natural-language query by summing up the term vectors of each term of the query. That is, $T_q = \sum_{i=1}^{m} t_i = \sum_{i=1}^{m}(f_{i1}, f_{i2}, ..., f_{ij}, ..., f_{in})$, where m is the total number of terms in the query and n is the number of semantic features. For example, the query "How can I play sound using Clip in Java?" is normalized to ("plai", "sound", "us", "clip") that has four terms. Then we obtain the vector representation of this query by summing up the term vectors of each of the four terms "plai", "sound", "us" and "clip".

2) Inferring relevant API class-names for the query. Similarly, we obtain the term vector representation of each API class-name from term vectors that are generated from the training data. We denote the vector representation of an API class-name as $T_c = (f_1, f_2, ..., f_i, ..., f_n)$, where n is the number of semantic features.

For a natural-language query q, we compute its semantic distance to each API class-name c. The semantic distance is measured by the cosine similarity [29] between their vector representations, and can be computed using Eq. (1).

$$D_{q,c} = \frac{T_q \cdot T_c}{|T_q||T_c|} \tag{1}$$

where T_q is the vector representation of the natural-language query q, and T_c is the vector representation of the API class-name c.

Then we rank the API class-names based on their semantic distances to the query.

As the more popular an API is, the more widely the API class is used to accomplish a programming task. The popularity of the API class-names is another important feature to consider [19]. The popularity of an API class-name is measured by the number of Java files that use the corresponding API class in the corpus. Hence, we re-rank the top 10 API class-names based on their popularities.

Finally, we select the top k API class-names from the resulting ten API class-names to expand the query. After attempting different numbers (*i.e.*, $k = 1$, 3, 5, 7, 10) on a development set of five queries, we decide to set $k = 5$. The development set is only used for choosing k, and is not included in the evaluation step. In addition, the completeness and redundancy of the selected API class-names can be balanced when setting $k = 5$. Specifically, using only the top ranked API class-name (*i.e.*, $k = 1$) might be insufficient to help improve the search efficiency. However, recommending many API class-names (*e.g.*, $k = 10$) to a query would increase the chance to include redundant API class-names that are not the most relevant to the query. For the query "How can I play sound using Clip in Java?" (see Fig. 1), the top 5 selected API class-names are: "Player", "AudioInputStream", "LineUnavailableException", "Game", and "AudioClip".

3.4 Executing the Expanded Queries

We execute the expanded queries using a two-layer code search engine. The two layers are applied following the order: the vector space model (VSM) [22] and the weighted-sum ranking schema [12].

1) The vector space model is used to retrieve an initial set of relevant code examples for expanded queries. However, VSM is a search engine for general text retrieval and therefore, is not optimized for code search. Solely using the similarity between the code example and the query is insufficient to retrieve relevant code examples [24]. Therefore, based on the similarity scores of the retrieved code examples that are reported by VSM, we further perform the weighted-sum ranking schema.

2) The weighted-sum ranking schema computes the ranking score of a candidate code example using a weighted sum of various features of the code example [12]. In our approach, we consider the following five code example features:

(F1) f_v is the similarity score produced by an off-the-shelf VSM technique Lucene[6] to measure the similarity between two texts (*i.e.*, the query and the code example in our case). Therefore, f_v measures the text similarity between the query and the whole body of the code example.

(F2) f_s measures the cosine similarity between the query and the signature of the code example [19]. The signature of the code example only includes the method name and the type of each parameter. Thus f_s captures the relevance between the query and the method declaration (*i.e.*, method name and parameter types).

(F3) f_n denotes the sum of the total number of occurrences that each line of code in the code example appears in the corpus [5]. f_n describes the popularity of the code example.

(F4) f_p is the number of the parameters of the code example, and reflects the amount of information required to execute the code example.

(F5) f_a denotes whether the recommended API class-names for the query expansion appear in the parameters of the code example. If the answer is yes, f_a is set to -1, otherwise, it is 0.

We compute the ranking score of a code example as the weighted sum of its five features, using Eq. (2).

$$S = wf_v + wf_s + 2wf_n + wf_p + wf_a \tag{2}$$

where f_v, f_s, f_n, f_p, f_a are the five features of a code example, and w is the weight for the features.

In our implementation, we assign equal weights to

6. Lucene: https://lucene.apache.org/

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TSE.2017.2750682, IEEE Transactions on Software Engineering

TABLE 3
Summary of our corpus.

Feature	Number
Number of projects	24,666
Number of Java files	2,882,451
Number of code snippets	23,677,216

each feature (except for f_n). We set the weight of feature f_n twice as large as other weights since ranking code examples based on feature f_n performs better than other features [19].

4 Case Study Setup

In this section, we present the setup of our case study to evaluate the performance of our proposed approach. The case study setup includes our corpus of code snippets, the golden set, and baseline approaches for comparison purpose.

4.1 Corpus

A large-scale corpus of code snippets is needed for evaluating the performance of retrieving code examples for a given code search query. To build the corpus, we download a big inter-project repository IJaDataset[7] that was created by Keivanloo et al. [19] and later was used by Svajlenko et al. [34] to build a benchmark for clone detection. The dataset contains 24,666 open-source projects that are collected from SourceForge and Google Code. In total, there are 2,882,451 unique Java files.

To extract code snippets from Java files, we use the same approach as the earlier work [26] on code search and recommendation of Java source code. Specifically, we use a Java syntax parser[8] available in Eclipse JDT to extract one code snippet from each single method in the Java files. Finally, 23,677,216 code snippets are extracted and added into the code snippet corpus. Table 3 summarizes the key descriptive statistics of the corpus.

4.2 Golden set

A golden set is a set of queries associated with a list of code examples that are correctly tagged to be relevant to each query. A golden set is essential to evaluate how our approach performs in recommending code examples for natural-language code search queries.

1) First golden set. The first golden set is obtained from a benchmark that was created for the clone detection purpose by Svajlenko et al. [34]. In the benchmark, there are thousands of code snippets from ten distinct functionalities (hereby, target functionalities) that are frequently used in open-source Java projects. The number of target functionalities has been expanded to 40 in the new version of the benchmark dataset[9]. In the benchmark, each code snippet is tagged whether it correctly implements each target functionality. The tagging process has been done by three different annotators in their study [34].

```
1  public static void copyFile(File srcFile, File destFile)
2          throws IOException {
3      if (!destFile.exists()) {
4          destFile.createNewFile();
5      }
6
7      FileInputStream fis = new FileInputStream(srcFile);
8      FileOutputStream fos = new FileOutputStream(destFile);
9
10     FileChannel source = fis.getChannel();
11     FileChannel destination = fos.getChannel();
12
13     destination.transferFrom(source, 0, source.size());
14
15     source.close();
16     destination.close();
17
18     fis.close();
19     fos.close();
20 }
```

Fig. 3. One code example in the first golden set.

For each target functionality, we locate its corresponding natural-language code search query from SO. The benchmark contains the specification of how to implement each target functionality. To find the corresponding natural-language query for each target functionality from SO, we check whether the accepted answer of the SO question exactly matches its specification.

For each target functionality, we consider a code snippet is truly relevant to the corresponding query if it is tagged to correctly implement the target functionality. For example, one code example in the golden set, as shown in Fig. 3, correctly implements the target functionality "Copy File".

Among the 40 target functionalities, we filter the ones that do not use APIs for implementing the functionality. This is because our approach expands the initial queries with API class-names, and it would make no difference on the search performance if APIs are not used. The authors manually reviewed the 40 target functionalities in the benchmark [34] and finally obtained 24 target functionalities. The corresponding queries and code examples of the 24 target functionalities are then selected to evaluate the performance of our approach.

2) Second golden set. To evaluate search engines, the minimum number of queries in a golden set is 50 [22]. We further created a second golden set using 50 code search questions on Stack Overflow. In the second golden set, the correct code examples for each query are identified using the pooling method [22]. The pooling method is the most standard evaluation method in information retrieval and was previously

7. IJaDataset: https://github.com/clonebench/BigCloneBench
8. The parser from Eclipse JDT: http://www.eclipse.org/jdt/
9. Benchmark dataset: https://github.com/XBWer/BigCloneBench

chosen to evaluate the schema for code example retrieval [2]. In the pooling method, evaluators assess the relevance of the top k results returned by a number of different approaches instead of the entire list of returned results. The pooling method is usually applied if the corpus is very large. Our corpus contains 23,677,216 code snippets in total; therefore it is impossible to manually examine all retrieved code snippets for each query.

For each query, we put the top k code snippets that are retrieved by six studied approaches (see Fig. 4 and 6) into a pool, and remove any duplicates. Annotators are provided the entire set of code snippets from the pool in a randomized order. Then Annotators independently label the relevance of each code snippet. We conjecture that, in practice, developers would have changed their queries if they could not find relevant code snippets on the first five pages. Most search engines (*e.g.*, Google) always present only ten results on each page [2]. So we choose to use the top 50 results (*i.e.*, $k = 50$, five pages with 10 results per page) to build the pool in our evaluation. Finally, we get a pool with at most 300 code snippets for each query.

The creation of the second golden set has been done by three annotators, who are graduate students with more than five years' experience in Java development. We apply Cohen's kappa coefficient to measure inter-annotator agreement. As we have three annotators, we compute the pair-wise Cohen's kappa coefficient and calculate the average value. In our study, the average Cohen's kappa coefficient is 0.80, indicating a good agreement [22]. When there is a disagreement, the majority rule is applied.

We combine the first and the second golden set for evaluation. The combination contains 74 natural-language code search queries in total. All the 74 queries can be found in the replication package.

4.3 Comparison Baselines

In the text retrieval literature, a variety of approaches for query expansion have been proposed [4]. When applying in software engineering data, three approaches (*i.e.*, *Dice* [8], *Rocchio* [28], and *RSV* [45]) are demonstrated to perform the best [14]. Therefore, we choose the three approaches as our baselines to compare our approach with.

In the three approaches, the candidate terms are selected from the top k code examples that are ranked by a ranking strategy. Among all candidate terms, the top n terms are recommended to expand the initial query. Similar to the study in Haiduc et al. [14], we set $k = 5$, and $n = 10$. The details of the three approaches are described as follows.

1) Dice. This approach ranks the terms in the top k code examples based on their *Dice* similarity to the terms of the initial query. The *Dice* similarity [8] can be computed as:

$$Dice = \frac{2df_{q \cap s}}{df_q + df_s} \tag{3}$$

where q denotes a term contained in a query; s is a term extracted from the top k code examples; df_q, df_s and $df_{q \cap s}$ denote the number of code examples in the corpus that contain q, s, or both q and s, respectively [8].

2) Rocchio. This approach orders the terms in the top k code examples using Rocchio's method [28]. The score of each term is computed as the sum of the $tf \times idf$ scores of the term in the top k code examples, as shown in Eq. (4).

$$Rocchio = \sum_{c \in R} tf(t, c) \times idf(t, c) \tag{4}$$

where R is the set of the top k code examples in the result list; c is a code example in R; and t is a term in the code example c [28].

The term frequency $tf(t, c)$ is the number of times of term t appearing in code example c. The term frequency $tf(t, c)$ indicates the importance of term t over other terms in code example c [22]. The inverse document frequency $idf(t, c)$ indicates the specificity of term t to code example c that contains it. The inverse document frequency $idf(t, c)$ can be considered as the inverse of the number of code examples in the corpus thats contain term t [22].

3) RSV. This approach uses the Robertson Selection Value (RSV) [45] to compute the ranking score for the terms in the top k code examples, using Eq. (5).

$$\begin{aligned} RSV = \sum_{c \in R} & tf(t, c) \times idf(t, c) \left[p(t|R) - p(t|C) \right] \\ & p(t|R) = freq(t \in R)/|R|_t \\ & p(t|C) = freq(t \in C)/|C|_t \end{aligned} \tag{5}$$

where C denotes the corpus of code snippets; R is the set of top k code examples; c is a code example in R; and t is a term in code example c; $freq(t \in R)$ and $freq(t \in C)$ are the number of times that term t appears in R and C, respectively; $|R|_t$ and $|C|_t$ are the number of terms in R and C, respectively [45].

4.4 Performance Measure

In our experiment, we consider three measures of the effectiveness of each code search approach:

1) Coverage that measures how good an approach is across a set of queries. Coverage of a code search approach is defined as the proportion of queries [2], for which there is at least one relevant code snippet among the top 10 retrieved results by the approach.

2) The rank of the first correct answer that measures how good an approach is for an individual query. The rank of the first correct answer is the position of the first correct answer (*i.e.*, truly relevant code example) in the result list.

3) Mean reciprocal rank (MRR) [39] is a statistic measure on the performance of a ranking approach,

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TSE.2017.2750682, IEEE Transactions on Software Engineering

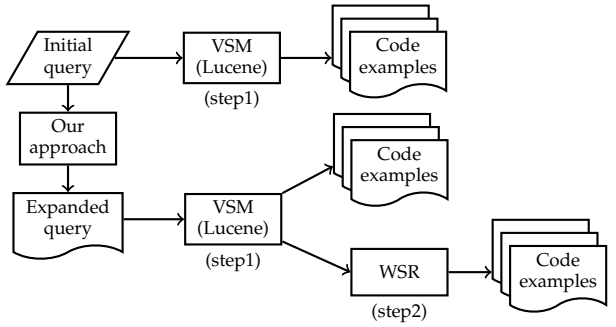

Fig. 4. The approach for studying the efficiency of our approach in code search (RQ3).

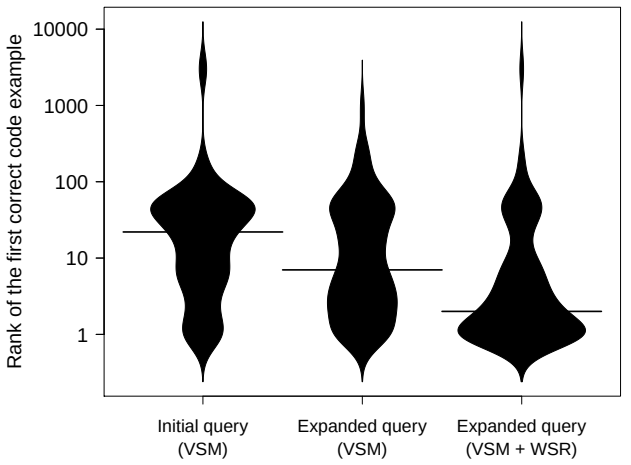

Fig. 5. The beanplots of the rank of the first relevant code example using the initial queries, the expanded queries without and with the weighted-sum ranking schema. The overall line depicts the median rank of the first relevant code example.

and is widely used in the information retrieval (IR) community. It is defined as the average of the reciprocal ranks (RR) of the first correct answer, using the equation $MRR = \frac{1}{|Q|}\sum_{i=1}^{|Q|} RR_i$, where $|Q|$ is the total number of queries used for evaluation and RR_i is the reciprocal rank of the first correct answer for the ith query (*i.e.*, $1/rank_i$).

5 Case Study Results

In this section, we discuss the results of our case study. For each research question, we describe the motivation, analysis approach, and findings.

RQ3: Does the expanded query work better than the natural-language query in code search?

Motivation. In **RQ2**, we confirm the importance of using identifiers to retrieve relevant code examples. Then we propose an approach to expand natural-language queries with API class-names that are semantically related to the initial query. In this question, we aim to evaluate whether our approach can effectively expand natural-language queries towards increasing the rank of the first relevant code example.
Approach. As our proposed approach contains two parts (*i.e.*, query expansion and the weighted-sum ranking schema), we design our experiment in two steps. The process is illustrated in Fig. 4.

1) Study the effect of expanding queries. We execute both the initial query and expanded query in Lucene (*i.e.*, a tool that implements VSM) and retrieve code examples from the corpus of code snippets. We compare the search results of the initial and expanded queries to evaluate the performance of query expansion. We calculate coverage and the rank of the first relevant code example, and test the following null hypothesis:

H_{30}^{1a}: there is no difference between the ranks of the first relevant code example returned by executing the initial queries and the ones returned by executing the expanded queries.

To test the null hypothesis H_{30}^{1a}, we apply the Wilcoxon signed-rank test with the 95% confidence level (*i.e.*, p-value < 0.05). Wilcoxon signed-rank test has no assumption about the distribution of the assessed variable [30], and it has been commonly applied in the information retrieval (IR) literature to compare IR measures [38, 46]. If p-value is less than 0.05, we reject the null hypothesis H_{30}^{1a} and conclude that the search performance of using the initial and expanded queries is significantly different. We further compare the ranks of the first relevant code example to find which approach performs better.

The search performance is improved if the MRR value increases. To assess if the improvement in terms of the MRR value is statistically significant, we conduct the Wilcoxon signed-rank test with the 95% confidence level (*i.e.*, p-value< 0.05) to test the null hypothesis:

H_{30}^{1b}: there is no difference in the reciprocal ranks (RR) between using the initial queries and using the expanded queries.

2) Study the effect of using the weighted-sum ranking schema. We compare the search results of the expanded query with and without the weighted-sum ranking schema (*i.e.*, WSR). This helps us understand the impact of the weighted-sum ranking schema on the search performance. Similarly, we compute coverage, the rank of the first relevant code example, and the MRR values. We test the following null hypothesis using the Wilcoxon signed-rank test with the 95% confidence level (*i.e.*, p-value < 0.05):

H_{30}^{2a}: there is no difference between the ranks of the first relevant code example returned by executing the expanded query without the weighted-sum ranking schema and the ones returned by executing the expanded query with the weighted-sum ranking schema.

H_{30}^{2b}: there is no difference in the reciprocal ranks (RR) with and without using the weighted-sum ranking schema.

Findings. The search performance for using the expanded query is better than using the initial query. By expanding the query, the coverage of code searches is increased from 36% to 54% for the 74 queries in the combination of our two golden sets. Fig. 5 depicts the rank of the first relevant code example in our three settings. The median rank of the first relevant code example is lifted up from 22 to 7 by expanding queries. As pointed out by Gottipati et al. [11], the higher rank a relevant code example has, the more valuable it is for a developer, because it is more likely to be examined by a developer. Moreover, the p-value of the Wilcoxon signed-rank test is 0.03. Therefore, we reject the null hypothesis H_{30}^{1a}. In addition, for 58% (*i.e.*, 43 out of 74) of queries, expanding the initial query yields higher ranks of the first relevant code example, and ties occur in seven queries.

The MRR values obtained using the initial queries and the expanded queries on the entire set of 74 queries are 0.20 and 0.29, respectively. The p-value of the Wilcoxon signed-rank test is 0.03. Hence, we reject the null hypothesis H_{30}^{1b}, and conclude that our query expansion can significantly improve the search performance in terms of the coverage, the rank of the first relevant code example, and the MRR value.

The weighted-sum ranking schema can further improve the search performance. With the weighted-sum ranking schema, the coverage of code searches is further increased from 54% to 76%, and the median rank of the first relevant code example is lifted up from 7 to 2. Our approach (*i.e.*, query expansion and the weighted-sum ranking schema) can retrieve relevant code examples in the top 2 code snippets for 54% (*i.e.*, 40 out of 74) of queries. The p-value of the Wilcoxon signed-rank test is $3.56e$-03. We reject the null hypothesis H_{30}^{2a}.

The MRR value obtained on the entire set of 74 queries is increased from 0.29 to 0.53, when using the weighted-sum ranking schema. The p-value of the Wilcoxon signed-rank test is $1.18e$-04. Hence, we reject the null hypothesis H_{30}^{2b}. We conclude that the weighted-sum ranking schema can further significantly improve the search performance in terms of the coverage, the rank of the first relevant code example, and the MRR value.

In summary, with query expansion, relevant code examples of 50% of the 74 studied queries can be retrieved in the top 7 results which usually appear on the first result page and are likely to be examined by developers. The weighted-sum ranking schema further saves the effort of developers by potentially providing relevant code examples in the top 2 results and increasing the coverage of code searches.

Our query expansion can significantly improve the search performance, and the weighted-sum ranking schema can bring a further improvement.

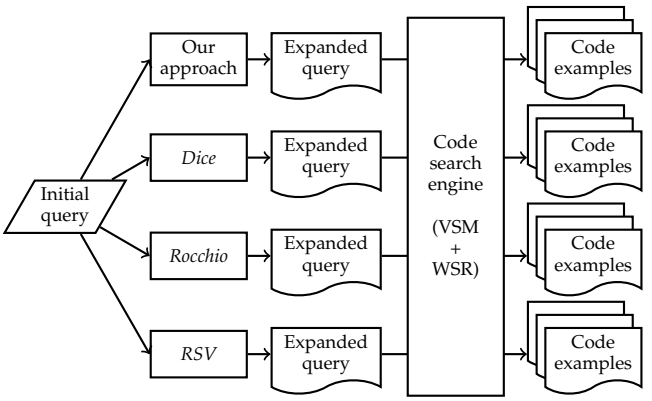

Fig. 6. The approach for comparing the four query expansion approaches (RQ4).

RQ4: Does our approach outperform the off-the-shelf query expansion approaches in answering natural-language code search queries?

Motivation. As aforementioned, three off-the-shelf approaches (*i.e.*, *Dice* [8], *Rocchio* [28], and *RSV* [45]) perform the best for software engineering data among a variety of query expansion approaches. To expand a query, the three approaches can generate related terms from the initial query. In this research question, we aim to study whether our expansion approach is better than the three popular approaches in the context of code search.

Approach. To address this question, we implemented the three query expansion approaches as described in Section 4.3. As presented in Fig. 6, we first expand the initial query using each of the four approaches. For each of the four expanded queries, we apply it in Lucene (*i.e.*, VSM) to retrieve the top 10 code examples. Then we apply the weighted-sum ranking schema (*i.e.*, WSR) to obtain the final rank of the ten code examples for each of the four approaches.

We compare the efficiency of our query expansion approach and the three approaches. For each off-the-shelf approach, we test the following null hypothesis, using the Wilcoxon signed-rank test with the 95% confidence level (*i.e.*, p-value < 0.05):

H_{40}^{a}: there is no difference between the ranks of the first relevant code example obtained for queries expanded by our approach and the ones obtained for queries expanded by the off-the-shelf approach.

If p-value is less than 0.05, we reject the null hypothesis H_{40}^{a} and conclude that the efficiency of our query expansion approach and the off-the-shelf approach is significantly different in terms of the rank of the first relevant code example. We further compare the ranks of the first relevant code example to find which query expansion approach performs better.

To assess if the improvement in the MRR value is statistically significant, we conduct the Wilcoxon signed-rank test with the 95% confidence level (*i.e.*, p-value< 0.05) to test the null hypothesis:

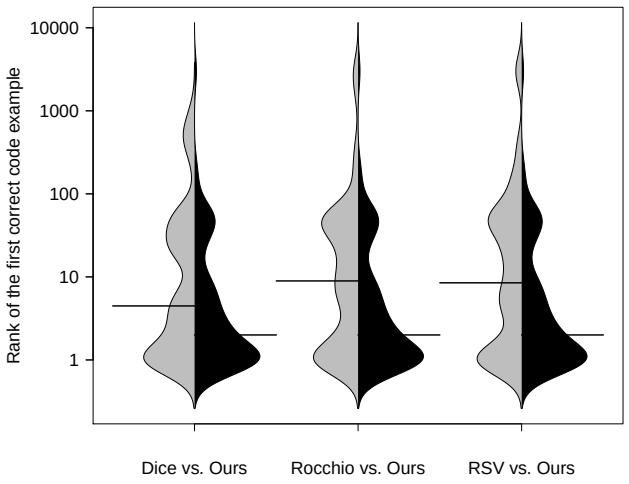

Fig. 7. The beanplots of the rank of the first relevant code example using the three off-the-shelf approaches and our approach. The overall line depicts the median rank of the first relevant code example.

H_{40}^{b}: there is no difference in the reciprocal ranks (RR) between using the initial queries and using the expanded queries.

Findings. Our approach performs better than the three off-the-shelf approaches on expanding natural-language queries for code search. The coverage of successful code searches is 76% using our approach, which is greater than the three off-the-shelf approaches that achieve 59% (*Dice*), 54% (*Rocchio*) and 51% (*RSV*). Fig. 7 presents the comparison result between the three off-the-shelf approaches and our approach using the combination of the two golden sets. The grey side represents the performance for the queries expanded by each of the three off-the-shelf approaches, while the black side denotes the performance for the queries expanded by our approach.

The median rank of the first relevant code example when using the queries expanded by our approach is 2. When using the three off-the-shelf approaches, the average rank of the first relevant code example is 4.5 (*Dice*), 9 (*Rocchio*) and 8.5 (*RSV*), respectively. The p-value of the Wilcoxon signed-rank test is 8.69e-03 (*Dice*), 4.62e-03 (*Rocchio*) and 1.51e-04 (*RSV*), respectively. Therefore, we reject the null hypothesis H_{40}^{a} for each off-the-shelf approach. Comparing to *Dice*, *Rocchio* and *RSV*, our approach results in better ranks of the first relevant code example for 47% (35/74), 50% (37/74), and 54% (40/74) of queries, respectively. Our approach exhibits ties with *Dice*, *Rocchio* and *RSV* for 30%(22/74), 24%(18/74), and 26%(19/74) of queries, respectively.

The MRR values obtained on the entire set of 74 queries for *Dice*, *Rocchio*, *RSV*, and our approach are 0.42, 0.36, 0.36, and 0.53, respectively. The p-values of the Wilcoxon signed-rank test are 0.03 (comparing to *Dice*), 4.18e-03 (comparing to *Rocchio*), and 2.10e-03 (comparing to *RSV*), respectively. Hence, we reject the null hypothesis H_{30}^{2b} in all cases.

In summary, we conclude that our approach outperforms the studied off-the-shelf query expansion approaches in expanding natural-language code search queries in terms of the coverage, the rank of the first relevant code example, and the MRR value.

> *On query expansion, our approach outperforms the studied three off-the-shelf approaches in terms of coverage and the rank of the first relevant code example.*

6 Threats to Validity

In this section, we discuss the threats to validity that might affect the results of the study by following the common guidelines provided by Yin [44].

Threats to construct validity concern whether the setup and measurement in the study reflect real-world situations. The threats to construct validity of our study mainly come from the categorization of Stack Overflow questions, the selection of code search queries and the creation of the golden sets. For the classification of the SO questions, we refer to the question types defined in the study by Stolee et al. [33]. Although we do not conduct cross validation in the classification process, the result is inline with their observation. We create two golden sets for the evaluation. One golden set is created based on the benchmark data used for clone detection study [34]. The other one is created using the pooling method, which was previously used by Bajracharya et al. [2] to evaluate code example retrieval schemes. The creation process of the two golden sets has been done by different judges, and the majority rule is applied when there is disagreement on the correctness of code examples. The usage of only six approaches is another threat to construct validity, as the pooling method used in the creation of the second golden set requires many approaches. However, we mitigate this threat by judging a large pool of documents.

Threats to internal validity concern the co-factors that may affect the experimental results. For our study, the main threats to internal validity are the number of recommended API class-names and the clicking analysis on the Koders data. We have tried different numbers of API class-names recommended to expand the natural-language code search queries using a small set of queries that are only used for parameter setting. We find that expanding the queries with the top five API class-names performs better than others, thus we set the number of recommended API class-names to five in our experiments. In RQ2, we decide whether the results returned by Koders answer users' queries using clicking analysis. It could be argued that users might just preview the search results without actually clicking into the results. However, as the result page of Koders shows only the top few lines

of each returned result, we conjecture that users are likely to click into the result if they find the relevant one to see more details.

Threats to external validity concern the generalization of the experiment results. The main threats to external validity of our study are the representativeness of our corpus and code search queries. Our corpus contains 23,677,216 code snippets extracted from 24,666 open-source projects. We select 74 descriptive questions from SO as natural-language code search queries to study the search effectiveness of our approach. The size of our query set is larger than the similar studies on code search and recommendation, *e.g.*, comparing to 22 queries used in [35], and 20 queries evaluated in [2]. In addition, the size of our query set meets the minimum number of queries recommended for search engine evaluation [22].

Threats to conclusion validity concern the relationship between treatment and outcome. We use a non-parametric statistic test (Wilcoxon signed-rank test) to show statistical significance of the obtained experiment results. Wilcoxon signed-rank test does not hold assumption on the distribution of data [30]. In addition, we consider that a code example contains only one method. Therefore, our conclusion on the efficiency of our approach is limited to retrieve code examples that have only one method.

Threats to reliability validity concern the possibility of replicating the study. We provide the necessary details needed to replicate our work. Replication package is publicly available:

http://www.feng-zhang.com/replications/TSEqueryExpansion.html

7 Related Work

In this section, we review the studies related to code search engines and query expansion technology.

7.1 Code Search

To solve programming tasks, developers can retrieve code snippets from code search engines by providing a query. Many approaches for code search and recommendation have been proposed (*e.g.*, [6, 16, 26, 33, 35, 40]). For instance, GitHub and Koders are two internet-scale code search engines that behave as general-purpose search engines (*i.e.*, accepting keywords as the input query).

Besides the keyword-based query, there are various forms of queries that are specialized for code search. Based on the query in the form "Source object type $\rightarrow$ Destination object type", ParseWeb [35] recommends code examples containing the given object type. Strathcona [16] locates relevant code examples through structural matching. The structural context of the code under development, including the method name being written and its containing class, is automatically extracted as the query. Wang et al. [42] propose to utilize dependency conditions among identifiers (*e.g.*, class and method names) to retrieve relevant code snippets within a project. Wang et al. [40] further combine topic analysis with dependency conditions to improve the accuracy of code search. Stolee et al. [33] design a semantic-based code search system that aims to generate specific functions to meet developers' specifications. Developers need to specify their query as precisely as possible, using a combination of keywords (*e.g.*, class or method signatures) and dynamic specifications (*e.g.*, test cases and security constraints). Mishne et al. [26] propose to use a sequence of API calls as queries to retrieve partial code snippets.

Different from aforementioned approaches, we use the general form of query (*i.e.*, a query described in natural language), but propose to expand the query with semantically related API class-names.

7.2 Query Expansion

Query expansion has long been an effective way to improve the performance of text retrieval (TR) engines [28]. The query expansion approaches belong to two major categories: interactive approaches and automatic approaches [14].

Interactive approach is based on relevance feedback and uses the "best" terms from the results of the initial query to augment the query for a further retrieval. The relevance feedback can require the involvement of users (*i.e.*, explicit relevance feedback, ERF) or can be done automatically (*i.e.*, pseudo/blind relevance feedback, PRF). ERF requires users to provide his/her judgements explicitly on the relevances of the returned documents to the query (*e.g.*, [9, 15, 41]). Although ERF can provide the most accurate feedback, its use in source code retrieval is highly limited as it demands users' numerous efforts in marking the relevant and irrelevant documents. On the other hand, PRF does not require user involvement. PRF uses the top set of the retrieved documents as an approximation to the set of relevant documents (*e.g.*, [14]). Therefore, it can only work when the initial query is reasonably strong (*i.e.*, at least some of the relevant documents can be retrieved).

Automatic approach is based on the automated identification of words that are semantically related to the initial query. Sridhara et al. [32] report that the words that are semantically related in English might be not semantically related in software products. Marcus et al. [23] use Latent Semantic Indexing (LSI) to determine the words that are more likely to co-occur in source code, and use the information of co-occurrence to recommend words that are semantically related to the queries. Ge et al. [10] implement pre-recommendation and post-recommendation for initial queries mainly by identifying frequently co-occurring terms techniques and synonym terms, respectively.

Yang and Tan [43] infer semantically related words through a pairwise comparison between the code segments and the comments in the source code. Tian et al. [36] propose to apply WordNet to find semantically related words. Similarly, Lu et al. [21] expand queries based on word similarity learned from WordNet. Moreover, Sisman and Kak [31] use positional proximity to identify the terms that are "close" to the initial query terms in the source code.

In this paper, we propose to use only API class-names for query expansion. The semantically related API class-names of each query are automatically identified by a neural network natural language model (specifically CBOW [25]).

8 Conclusion

A common practice of developers to solve programming problems is to ask questions on online forums (*e.g.*, Stack Overflow). We observe that questions aiming for code examples are prevalent on Stack Overflow. We consider such questions as natural-language queries to be fed into code search engines. It can relieve developers' programming burden to a great extent, if code search engines can return relevant code examples. However, code search engines usually perform well for queries with full identifiers, but not for natural-language queries that may not contain identifiers. Therefore, it is of significant interest to expand natural-language queries using related identifiers.

To this end, we propose an automated approach that identifies semantically related API class-names to a given natural-language query. First, we apply the continuous bag-of-words model (CBOW) [25] to learn vector representations of API class-names. Second, we compute the semantic distance between the initial query and API class-names using the learned term vectors. Then, we select the most semantically related API class-names to expand the initial query.

Our approach is evaluated using 74 natural-language queries. We measure the search efficiency using the coverage and the rank of the first desired code example in the retrieved list of code examples. The result of the case study shows that using our expanded queries are significantly better than using the initial queries. Using the weighted-sum ranking schema further improves the coverage of code searches. We also compare our query expansion approach with three popular query expansion approaches (*i.e.*, *Dice* [8], *Rocchio* [28], and *RSV* [45]) that have been demonstrated to perform the best for software engineering data [14]. The results show that our approach performs better than the three query expansion approaches. In summary, it is a promising direction to expand natural-language queries with API class-names.

In future, we plan to integrate our approach with a state-of-the-art code search engine to effectively recommend code examples for natural-language queries. It would be fruitful for existing code search engines to adopt our approach to enhance their support of natural-language queries. Moreover, we plan to retrieve more complex code examples that span several methods.

References

[1] S. K. Bajracharya and C. V. Lopes. Analyzing and mining a code search engine usage log. *Empirical Software Engineering*, 17(4-5):424–466, 2012.

[2] S. K. Bajracharya, J. Ossher, and C. V. Lopes. Leveraging usage similarity for effective retrieval of examples in code repositories. In *8th ACM SIGSOFT international symposium on Foundations of software engineering*, pages 157–166, 2010.

[3] M. G. Bulmer. *Principles of Statistics*. Courier Corporation, 1967.

[4] C. Carpineto and G. Romano. A survey of automatic query expansion in information retrieval. *ACM Computing Surveys*, 44:1–56, 2012.

[5] S. Chatterjee, S. Juvekar, and K. Sen. Sniff: A search engine for java using free-form queries. In *12th International Conference on Fundamental Approaches to Software Engineering*, pages 385–400, 2009.

[6] R. Cox. Regular expression matching with a trigram index or how google code search worked. In *Online resource: http://swtch.com/ rsc/regexp/regexp4.html*, 2012.

[7] CreativeResearchSystems. Sample Size Calculator. http://www.surveysystem.com/sscalc.htm, 2012. [Online; accessed 18-Feburary-2016].

[8] L. R. Dice. Measures of the amount of ecologic association between species. *Ecology*, 26:297–302, 2012.

[9] G. Gay, S. Haiduc, A. Marcus, and T. Menzies. On the use of relevance feedback in IR-based concept location. In *International Conference on Software Maintenance*, pages 997–1016, 2009.

[10] X. Ge, D. C. Shepherd, K. Damevski, and E. R. Murphy-Hill. How developers use multi-recommendation system in local code search. In *International conference on Visual Language and Human-Centric Computing*, pages 69–76, 2014.

[11] S. Gottipati, D. Lo, and J. Jiang. Finding relevant answers in software forums. In *26th International Conference on Automated Software Engineering*, pages 323–332, 2011.

[12] M. Grechanik, C. Fu, Q. Xie, C. McMillan, D. Poshyvanyk, and C. Cumby. Exemplar: A

source code search engine for finding highly relevant applications. *IEEE Transactions on Software Engineering*, 38:1069–1087, 2012.

[13] P. E. Greenwood and M. S. Nikulin. *A Guide to Chi-Squared Testing*. John Wiley & Sons, Inc., 1996.

[14] S. Haiduc, G. bavota, A. Marcus, R. Oliveto, A. D. Lucia, and T. menzies. Automatic query reformulations for textual retireval in software engineeing. In *International Conference on Software Engineering*, pages 842–851, 2013.

[15] J. H. Hayes, A. Dekhtyar, and S. K. Sundaram. Advancing candidate link generation for requirements tracing: The study of methods. *IEEE Transactions on Software Engineering*, 32:4–19, 2006.

[16] R. Holmes and G. C. Murphy. Using structural context to recommend source code examples. In *27th International Conference on Software Engineering*, pages 117–125, 2005.

[17] Thorsten Joachims. Optimizing search engines using clickthrough data. In *Proceedings of the Eighth ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, KDD '02, pages 133–142, 2002.

[18] Thorsten Joachims, Laura Granka, Bing Pan, Helene Hembrooke, Filip Radlinski, and Geri Gay. Evaluating the accuracy of implicit feedback from clicks and query reformulations in web search. *ACM Transactions on Information Systems*, 25(2), April 2007.

[19] I. Keivanloo, J. Rilling, and Y. Zou. Spotting working code examples. In *36th International Conference on Software Engineering*, pages 664–675, 2014.

[20] A. Kuhn, S. Ducasse, and T. Girba. Semantic clustering: Identifying topics in source code. *Information and Software Technology*, 49:230–243, 2007.

[21] Meili Lu, X. Sun, S. Wang, D. Lo, and Yucong Duan. Query expansion via wordnet for effective code search. In *Software Analysis, Evolution and Reengineering (SANER), 2015 IEEE 22nd International Conference on*, pages 545–549, March 2015.

[22] Christopher D. Manning, Prabhakar Raghavan, and Hinrich Schütze. *Introduction to Information Retrieval*. Cambridge University Press, New York, NY, USA, 2008.

[23] A. Marcus, A. Sergeyev, V. Rajlich, and J. I. Maletic. An information retrieval approach to concept location in source code. In *Working Conference on Reverse Engineering*, pages 214–223, 2004.

[24] C. McMillan, D. Poshyvanyk, M. Grechanik, Q. 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*, 22, 2013.

[25] T. Mikolov, K. Chen, G. Corrado, and J. Dean. Efficient estimation of word representations in vector space. In *Workshop at ICLR*, 2013.

[26] A. Mishne, S. Shoham, and E. Yahav. Typestate-based semantic code search over partial programs. In *ACM international conference on Object oriented programming systems languages and applications*, pages 997–1016, 2012.

[27] Doug Newcomb. The next big os war is in your dashboard. *Wired*, 2012.

[28] J. J. Rocchio. The smart retrieval system - experiments in automatic document processing. *Relevance feedback in information retrieval*, pages 313–323, 1971.

[29] G. Salton, A. Wong, and C. S. Yang. A vector space model for automatic indexing. In *Communications of the ACM*, pages 613–620, 1975.

[30] D. J. Sheskin. *Handbook of Parametric and Nonparametric Statistical Procedures*. Chapman and Hall/CRC, 2007.

[31] B. Sisman and A. C. Kak. Assisting code search with automatic query reformulation for bug localization. In *10th IEEE Working Conference on Mining Software Repositories*, pages 309–318, 2013.

[32] G. Sridhara, E. Hill, L. Pollock, and K. V. Shanker. Identifying word relations in software: A comparative study of semantic similarity tools. In *International Conference on Program Comprehension*, pages 123–132, 2008.

[33] K. T. Stolee, S. Elbaum, and D. Dobos. Solving the search for source code. *ACM Transactions on Software Engineering and Methodology*, 23, 2014.

[34] J. Svajlenko, J. F. Islam, I. Keivanloo, C. K. Roy, and M. M. Mia. Towards a big data curated benchmark of inter-project code clones. In *30th International Conference on Software Maintenance and Evolution*, 2014.

[35] S. Thummalapenta and T. Xie. Parseweb: A programmer assistant for reusing open source code on the web. In *22nd International Conference on Automated Software Engineering*, pages 204–213, 2007.

[36] Yuan Tian, D. Lo, and J. 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*, pages 44–53, Feb 2014.

[37] TIOBE. TIOBE Index for March 2016. http://www.tiobe.com/tiobe_index, 2016. TIOBE Software BV. [Online; accessed 22-March-2016].

[38] Julián Urbano, Mónica Marrero, and Diego Martín. A comparison of the optimality of statistical significance tests for information retrieval evaluation. In *Proceedings of the 36th International ACM SIGIR Conference on Research and Development in Information Retrieval*, SIGIR '13, pages 925–928, 2013.

[39] Ellen M. Voorhees. The trec-8 question answering track report. In *Proceedings of the 8th Text REtrieval*

Conference, TREC '99, pages 77–82, 1999.

[40] S. Wang, D. Lo, and L. Jiang. Code search via topic-enriched dependence graph matching. In *Reverse Engineering (WCRE), 2011 18th Working Conference on*, pages 119–123, Oct 2011. doi: 10.1109/WCRE.2011.69.

[41] Shaowei Wang, David Lo, and Lingxiao Jiang. Active code search: Incorporating user feedback to improve code search relevance. In *Proceedings of the 29th ACM/IEEE International Conference on Automated Software Engineering*, ASE '14, pages 677–682, New York, NY, USA, 2014. ACM. ISBN 978-1-4503-3013-8. doi: 10.1145/2642937.2642947.

[42] Xiaoyin Wang, David Lo, Jiefeng Cheng, Lu Zhang, Hong Mei, and Jeffrey Xu Yu. Matching dependence-related queries in the system dependence graph. In *Proceedings of the IEEE/ACM International Conference on Automated Software Engineering*, ASE '10, pages 457–466, 2010.

[43] J. Yang and L. Tan. Inferring semantically related words from software context. In *IEEE Working Conference on Mining Software Repositories (MSR)*, pages 161–170, 2012.

[44] Robert K. Yin. *Case Study Research: Design and Methods - Third Edition*. SAGE Publications, 3 edition, 2002.

[45] Hugo Zaragoza, Nick Craswell, Michael J. Taylor, Suchi Saria, and Stephen E. Robertson. Microsoft cambridge at TREC 13: Web and hard tracks. In *Proceedings of the Thirteenth Text REtrieval Conference, TREC 2004*, 2004.

[46] Justin Zobel. How reliable are the results of large-scale information retrieval experiments? In *Proceedings of the 21st Annual International ACM SIGIR Conference on Research and Development in Information Retrieval*, SIGIR '98, pages 307–314, 1998.

Feng Zhang is currently a postdoctoral research fellow with the Department of Electrical and Computer Engineering at Queen's University in Canada. He obtained his PhD degree in Computer Science from Queen's University in 2016. His research interests include empirical software engineering, mining software repositories, software analytics. His research has been published at several top-tier software engineering venues, such as the IEEE Transactions on Software Engineering (TSE), the International Conference on Software Engineering (ICSE), and the Springer Journal of Empirical Software Engineering (EMSE). More about Feng and his work is available at http://www.feng-zhang.com

Haoran Niu received the BEng degree from Harbin Institute of Technology, China, in 2013. She is currently working towards the M.A.Sc degree in the Department of Electrical and Computer Engineering at Queen's University, Canada. Her research interests are code search, code recommendation,and their applications in mobile application development.

Iman Keivanloo received his Ph.D degree in 2013 from Concordia University, Canada. He is currently a Post Doctoral Fellow in the Department of Electrical and Computer Engineering at Queen's University. His research interests include source code similarity search, clone detection, source code recommendation, and their applications for software evolution and maintenance.

Ying Zou is the Canada Research Chair in Software Evolution. She is an associate professor in the Department of Electrical and Computer Engineering, and cross-appointed to the School of Computing at Queen's University in Canada. She is a visiting scientist of IBM Centers for Advanced Studies, IBM Canada. Her research interests include software engineering, software reengineering, software reverse engineering, software maintenance, and service-oriented architecture. More about Ying and her work is available online at http://post.queensu.ca/~zouy