Empir Software Eng
https://doi.org/10.1007/s10664-017-9544-y

Augmenting and structuring user queries to support efficient free-form code search

Raphael Sirres[1] · Tegawendé F. Bissyandé[2] · Dongsun Kim[2] · David Lo[3] · Jacques Klein[2] · Kisub Kim[2] · Yves Le Traon[2]

Abstract Source code terms such as method names and variable types are often different from conceptual words mentioned in a search query. This vocabulary mismatch problem can make code search inefficient. In this paper, we present COde voCABUlary (CoCaBu), an approach to resolving the vocabulary mismatch problem when dealing with free-form

Communicated by: Denys Poshyvanyk

We make all our data available: source code of GitSearch, search indices, user study results. See https://github.com/serval-snt-uni-lu/cocabu. A prototype implementation of cocabu-based search engine, GitSearch, is live at http://www.cocabu.com.

✉ Dongsun Kim
dongsun.kim@uni.lu

Raphael Sirres
bconnectlu@gmail.com

Tegawendé F. Bissyandé
tegawende.bissyande@uni.lu

David Lo
davidlo@smu.edu.sg

Jacques Klein
jacques.klein@uni.lu

Kisub Kim
kisub.kim@uni.lu

Yves Le Traon
yves.letraon@uni.lu

[1] National Library of Luxembourg, 37, Boulevard F.D., Roosevelt 2450, Luxembourg

[2] Interdisciplinary Centre for Security, Reliability and Trust, University of Luxembourg, 29, Avenue J.F, Kennedy 1855, Luxembourg

[3] School of Information Systems, Singapore Management University, 80 Stamford Road, Singapore 178902, Singapore

Published online: 26 January 2018

code search queries. Our approach leverages common developer questions and the associated expert answers to augment user queries with the relevant, but missing, structural code entities in order to improve the performance of matching relevant code examples within large code repositories. To instantiate this approach, we build GITSEARCH, a code search engine, on top of `GitHub` and `Stack Overflow` Q&A data. We evaluate GITSEARCH in several dimensions to demonstrate that (1) its code search results are correct with respect to user-accepted answers; (2) the results are qualitatively better than those of existing Internet-scale *code search engines*; (3) our engine is competitive against *web search engines*, such as Google, in helping users solve programming tasks; and (4) GITSEARCH provides code examples that are acceptable or interesting to the community as answers for `Stack Overflow` questions.

Keywords Code search · GitHub · Free-form search · Query augmentation · StackOverflow · Vocabulary mismatch

1 Introduction

Code search is an important activity in software development since developers are regularly searching (Sadowski et al. 2015) for code examples dealing with diverse programming concepts, APIs, and specific platform peculiarities. Such examples can indeed help them practice programming against a library and platform, or they can immediately be used for inspiration in software development tasks. Because contemporary programmers often implement most of the program elements (e.g., classes and methods) based on existing programs already written by other programmers (McMillan et al. 2012), an effective code search engine is a critical factor for programming productivity.

Open source project hosting platforms, such as `GitHub`, SourceForge, and BitBucket now offer an opportunity for students, researchers and developers to access real-world software projects for improving their work. It is, however, challenging to locate relevant source code due to the enormous size of existing code repositories. For instance, as of August 2015, `GitHub` is hosting more than 25 millions private and public code repositories.[1] To help developers search for source code, several Internet-scale code search engines (Gallardo-Valencia and Elliott Sim 2009), such as Openhub (2016) and Codota (2016) have been proposed. The advantage of these engines is that users can express their queries in a list of keywords (i.e., free-form queries) rather than specific program elements such as API classes and methods.

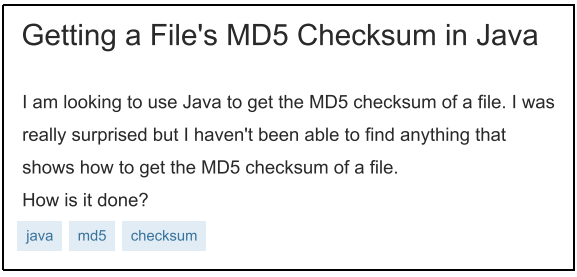
Unfortunately, these Internet-scale code search engines have an accuracy issue since they treat source code as natural language documents. Source code, however, is written in a programming language while query terms are typically expressed in natural language. As a result, searching source code with query keywords in natural language often leads to irrelevant and low-quality search results unless the keywords exactly correspond to program elements. According to Hoffmann et al. (2007), however, around 64% of programmer web queries for code are merely descriptive but do not contain actual names of APIs, packages, types, etc.

As in any search engine, the terms in a code search query must be mapped with an index built from the code. Unfortunately, the construction of such an index as well as the mapping

[1] https://github.com/about/press (verified 14.08.2015).

process are challenging since “no single word can be chosen to describe a programming concept in the best way” (Furnas et al. 1987). This is known in the literature as the vocabulary mismatch problem: user search queries frequently mismatch a majority of the relevant documents (Furnas et al. 1987; Zhao and Callan 2010, 2012; haiduc et al. 2013). This problem occurs in various software engineering research work such as retrieving regulatory codes in product requirement specifications (Cleland-Huang et al. 2010), identifying bug files based on bug reports (Nguyen et al. 2011), and searching code examples (Haiduc et al. 2013; Haiduc et al. 2013; Hill et al. 2014).

The vocabulary mismatch problem is further exacerbated in code search engines where the source code may be poorly documented or may use non-explicit names for variables and method names (Kim and Kim 2016). To work around the translation issue between the query terms and the relevant code, one can leverage a developer community. Actually, developers often resort to web-based resources such as blogs, tutorial pages, and Q&A sites. `Stack Overflow` is one of such leading discussion platforms, which has gained popularity among software developers. In `Stack Overflow`, an answer to a question is typically short texts accompanied by code snippets that demonstrate a solution to a given development task or the usage of a particular functionality in a library or framework. `Stack Overflow` provides social mechanisms to assess and improve the quality of posts that leads implicitly to high-quality source code snippets. Figure 1 shows an example of the vocabulary mismatch problem.



a Example of questions in `Stack Overflow`.

There's an input stream decorator, `java.security.DigestInputStream`, so that you can compute the digest while using the input stream as you normally would, instead of having to make an extra pass over the data.

```
MessageDigest md = MessageDigest.getInstance("MD5");
try (InputStream is = Files.newInputStream(Paths.get("file.txt"));
     DigestInputStream dis = new DigestInputStream(is, md))
{
  /* Read decorated stream (dis) to EOF as normal... */
}
byte[] digest = md.digest();
```

b Potential solution for the question in (a).

Fig. 1 Example of the vocabulary mismatch problem. Regarding the question shown in (**a**), a user of `Stack Overflow` posted a potential answer as shown in (**b**). The keywords of the question include `MD5, checksum, Java, file`. However, the snippet in the answer contains a different set of keywords such as `MessageDigest, InputStream, DigestInputStream, digest, MD5, file`. Using the keywords of the question when searching for code examples has significantly low possibility to locate code fragments similar to the snippet in the answer

While code snippets found in Q&A sites certainly accelerate the software development process, they fail to explore the potential of large code repositories. Typically, those code snippets are manually crafted by developers rather than being actual examples from source code repositories. Thus, snippets often omit context information (e.g., variable types and initialization values) that might be necessary to understand interactions with other relevant components. On the other hand, actual examples in source code repositories can provide different views on how a single functionality can be implemented by different APIs. Source code repositories also contain concrete code that demonstrates the interaction between various modules and APIs of interest. Besides, usually, in Q&A sites, an acceptable answer only exists when the question, or a very similar one, has been asked before. Otherwise, the questioner must wait for other experienced developers to provide answers.

Our work focuses on building an approach to automatically expand developer code search queries. Specifically, we aim at translating free-form queries to augment them with relevant program elements. To augment a user query, we consider first finding similar (in terms of natural language words) queries for which we have some sketched answers. Then we can collect from these answers some important code keywords. Finally, such code keywords are simply used to enrich the user's initial free-form terms. This query expansion is effective in retrieving relevant code search results even when the user has not provided in his query terms essential information such as API names.

Contributions We propose a novel approach to augmenting user queries in a free-form code search scenario. This approach aims at improving the quality of code examples returned by Internet-scale code search engines by building a COde voCABUlary (CoCaBu). The originality of CoCaBu is that it addresses the vocabulary mismatch problem, by expanding/enriching/re-targeting a user's free-form query, building on similar questions in Q&A sites so that a code search engine can find highly relevant code in source code repositories.

Overall, this paper makes the following contributions:

- CoCaBu **approach to the vocabulary mismatch problem:** We propose a technique for finding relevant code with free-form query terms that describe programming tasks, with no a-priori knowledge on the API keywords to search for. In this regard, we differ from several state-of-the-art techniques, which perform by searching relevant usage examples of APIs that the user can already list as relevant for his task (Moreno et al. 2015; Keivanloo et al. 2014; Mandelin et al. 2005; Chatterjee et al. 2009).
- GitSearch **free-form search engine for GitHub:** We instantiate the CoCaBu approach based on indices of Java files built from `GitHub` and Q&A posts from `Stack` Overflow to find the most relevant source code examples for developer queries.
- **Empirical user evaluation**: We present the evaluation results implying that GitSearch accurately extends user queries to produce correct (i.e., relevant) results. Comparison with popular code search engines further shows that GitSearch is more effective in returning acceptable code search results. In addition, Comparison against web search engines indicates that GitSearch is a competitive alternative. Finally, via a live study, we show that users on Q&A sites may find GitSearch's real code examples acceptable as answers to developer questions.

The remainder of this paper is organized as follows. Section 2 motivates our work further, listing some limitations in the state-of-the-art and introducing the key ideas behind our approach. Section 3 then overviews the CoCaBu approach. We provide evaluation results in Section 5 and discuss related work in Section 6. Finally, Section 7 concludes the paper.

2 Motivation

The literature contains a large body of approaches that attempt to solve the vocabulary mismatch problem. They either 1) use a *controlled vocabulary* (Liu et al. 1999) maintained by experts in specific and restricted domains; or 2) automatically derive a *thesaurus* (Eckert et al. 2007), e.g., word co-occurrence statistics in an exhaustive corpus; or 3) *interactively expand* user queries (Ruthven 2003), e.g., by recommending other terms from previous query logs; or 4) *automatically expand* queries (Carpineto et al. 2001) by adding derived words from the terms included in the original query, e.g., add the `integer` word in a query with `int`; or 5) completely *rewrite* the query automatically (Gollapudi et al. 2011). Most of these approaches are not suitable in the settings of a code search engine, since i) the domain is not restricted, ii) the corpus is not finite, iii) query logs are not always available, iv) code terms and query terms may not share any stem words, and v) query terms remain valuable to be matched against identifiers in the code.

Furthermore in practice, implementing a *code* search engine has its own additional tasks: (1) relevant data is hidden in the deep web and unlinked; (2) the variety of concepts in programming languages, APIs, platforms or development environment challenges indexing; (3) the vocabulary mismatch problem complicates query processing; and (4) granularity of search output (e.g., code snippets, files, or applications) is also challenging to determine and satisfy.

Among the above tasks, query processing is one of the key components since search engine must match the query terms with relevant keywords from the index. The indexing step itself can improve speed and performance in finding relevant documents (source code files in our case) corresponding to a given search query. It often uses the salient keywords in a document. In code search, however, such keywords may not include API names since a single programming concept can be translated and implemented by several different classes and methods. This mismatch may degrade the quality of code search results.

2.1 Limitations of the State-of-the-art

Online code search engines such as Openhub (2016) and Codota (2016) perform basic string matching between user free-form queries and the code (which is then strictly considered as a text document, with no distinction between code and documentation). This, however, produces very low-quality results since programming language terms do not always match natural language words (Stylos and Myers 2006).

Figure 2 shows an example of OpenHub's search results for the query "Generating random words in Java?".[2] This top result from the search engine is not relevant: the returned snippet is for a program that *randomly selects a word from an array of words* rather than generating random words. This inaccurate search result occurs because the words used in the query are not appropriate for direct match with source code terms; "random *string* in Java" is the correct terminology that would have matched a more relevant program. Following results from the search engine were found irrelevant as well. The described example shows the limitation of the current practice in face of the vocabulary mismatch problem.

Our goal is to resolve this vocabulary mismatch problem in order to allow code search engines to return highly relevant code snippets for user free-form queries. Indeed, if we can

[2]This is a real question asked by a user in this post: http://stackoverflow.com/questions/4951997/generating-random-words-in-java.

File: RecyclerTest.java Project: OHA-Android-2.2_r1.1

```
80      // Read in dictionary of words
81      mWords = new ArrayList<String>(98568);      // count of words in words file
82      StringBuilder sb = new StringBuilder();
83      try {
84          Log.v(TAG, "Loading dictionary of words");
85          FileInputStream words = context.openFileInput("words");
102         Log.e(TAG, "can't open words file at /data/data/com.android.mms/files/words");
103         return;
104     }
105
106     // Read in list of recipients
107     mRecipients = new ArrayList<String>();
108     try {
109         Log.v(TAG, "Loading recipients");
110         FileInputStream recipients = context.openFileInput("recipients");
133     int wordsInMessage = mRandom.nextInt(9) + 1;   // up to 10 words in the message
134     StringBuilder msg = new StringBuilder();
135     for (int i = 0; i < wordsInMessage; i++) {
136         msg.append(mWords.get(mRandom.nextInt(mWordCount)) + " ");
137     }
```

Fig. 2 Top result provided by OpenHub for the free-form code search query "*Generating random words in Java?*"

appropriately transform words used in a search query to keywords found in source code, the search result would be more accurate as shown in Fig. 3; this is an actual search result of our approach described in Section 3. The produced code snippet, extracted from real world code, is practically identical to the manually crafted accepted answer for the question in the Q&A post.

Note that state-of-the-art approaches in the literature, such as Muse (Moreno et al. 2015) and MAPO (Xie and Pei 2006), focus on finding usage examples of API methods whose names must be explicitly indicated in the query. Thus, they may not be suitable for development tasks where users do not know the source code keywords of the relevant APIs. In particular, novice programmers may fail to get relevant code usage examples without knowing exactly necessary class or method names.

```
24  public class RandomString {
25
26      private static final char[] symbols = new char[36];
27      private static final Random random = new Random();
28
29      static {

37      {
38          char[] buf = new char[length];
39          for (int idx = 0; idx < length; ++idx)
40              buf[idx] = symbols[random.nextInt(symbols.length)];
41          return new String(buf);
42      }
```

Fig. 3 Top result provided by a CoCaBu-based search engine (see Section 3) for the same query used in Fig. 2. This code snippet was found in class `org.neo4j.vagrant.RandomString` of *simpsonjulian/neophyte* project from `GitHub`

Other techniques such as Sourcerer (Bajracharya et al. 2006) have proposed infrastructures to collect and model open source code data that users can query programmatically (e.g., SQL query statements). The Portfolio (McMillan et al. 2011) search engine returns output relevant functions and their usage scenarios. However, these approaches also simply match query terms with function names in the code base.

In summary, because of the vocabulary mismatch problem, current state-of-the-art approaches to code search fail to support entirely free-form and complex queries such as the ones developers are asking to other experienced developers on Q&A sites (cf. query in the caption of Fig. 2).

2.2 Key Intuition

Q&A posts contain a wealth of information that can be automatically leveraged by a code search engine. A typical Q&A post is a developer question accompanied with answers provided by experienced developers:

- In Q&A sites, developer questions, which are also often rewritten to make them explicit and limit the opportunities for duplicate questions, are good summaries of typical developer query terms.
- Code snippets embedded in experienced developer answers are a good starting point to systematically list relevant source code information related to developer question.

Thus, by leveraging developer questions from Q&A sites, and the associated code snippets, we can document concept mappings, i.e., the mappings between human concepts, which are expressed in questions, and program elements, which can be identified in code snippets. Once a large corpus of such mappings become available, the vocabulary mismatch problem can be alleviated. Indeed, any developer query, written in natural language, can be translated into a program query that explicitly makes references to specific program elements such as method and class names. This new query can then be directly matched against any source code file.

3 Our Approach

COCABU is about retrieving most relevant source code snippets to answer a free-form query given by a user. To resolve the vocabulary mismatch problem illustrated in Section 2.1, our approach leverages the intuition described in Section 2.2. Figure 4 provides an overview of our approach.

The search process begins with a free-form query from a user, i.e., a sentence written in a natural language:

(a) – For a given query, COCABU first searches for relevant posts in Q&A forums. The role of the Search Proxy is then to forward developer free-form queries to web search engines that can collect and rank entries in Q&A with the most relevant documents for the query.

(b) – COCABU then generates an augmented query based on the information in the relevant posts. To that end, it mainly leverages code snippets in the previously identified posts. Since these snippets are approved by developers as acceptable code examples from the posted question, COCABU can consider them translations of human concepts into program elements. COCABU's Code Query Generator then creates another

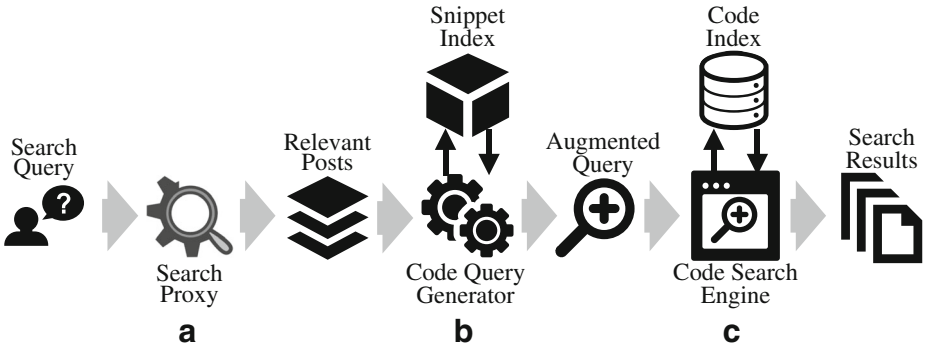


Fig. 4 Overview of CoCaBu

query which includes not only the initial user query terms but also program elements, such as method and class names, from the extracted snippets. To accelerate this step in the search process, CoCaBu builds upfront a snippet index for Q&A posts.

(c) – Once the augmented query is constructed, CoCaBu searches source code files for code locations that match the query terms. For this step, we can crawl a large number of public code repositories and build upfront a code index for program elements in the source code. It then leverages the code index to produce search results for a given augmented query. This search result can be presented to a user at different granularity level (e.g., relevant source code file, or code snippet).

To efficiently search source code in repositories for relevant code locations that match information from Q&A posts, CoCaBu makes indices of structural code entities in code snippets and source code files. Previous studies on code search and recommendation systems (Bajracharya 2010; Bajracharya et al. 2010; Lozano et al. 2011; Chen et al. 2012) have already proposed to take advantage of structural code information (e.g., method identifiers and class types) to improve query results. Indeed, if provided by user query, this information enables to map source code based on specific program elements. We use similar structural entities to those leveraged in many of previous work (Bajracharya 2010; Bajracharya et al. 2010; Lozano et al. 2011; Chen et al. 2012). Since structural code entities extraction process is specific to a programming language, we report such details in Section 4.2 when overviewing the GitSearch implementation case study.

The remainder of this section details the design of CoCaBu components (Sections 3.1–3.3).

3.1 Search Proxy

The search proxy takes a free-form query as an input and returns a set of relevant posts collected from developer Q&A sites as an output. The goal of this component is to collect sufficient data so that the search engine can later find out how natural language concepts can be translated into program elements. Indeed, code snippets in answers of Q&A posts can provide potential translation rules from concepts written in natural languages to program elements such as API methods or classes. As discussed in Section 2, such translation rules facilitate the subsequent code search process by alleviating the vocabulary mismatch problem that exists between user queries and source code elements.

Table 1 List of Q&A posts relevant to 'Generating random words in Java?'

Q&A site	Post title	Post ID
`Stack Overflow`	Generating random words of a certain length in java?	27429181
`Stack Overflow`	Random word from array list	20358980
`dummies.com`	How to Generate Words Randomly in Java	–
`java2notice.com`	How to create random string with random characters?	–
`coderanch.com`	Random string generation	374794

Relying on general purpose engines such as Google Web Search, Bing, and Yahoo Search, CoCaBu can search several different forums and rank the search results according to their relevancy to the query. Thus, in practice, once a user submits a code search query, the search proxy forwards it to a general-purpose web search engine to obtain related questions on the web. Since these search engines are specialized for text search, we assume that they are better than other built-in search engines in Q&A forums. Web search results are then filtered by the search proxy to eliminate URLs not related to Q&A posts. For example, if we want to consider only `Stack Overflow` posts, the search proxy would try to match the following pattern to collect relevant posts:

```
http://stackoverflow.com/questions/<ID>/<TITLE>
```

The ranking of relevant posts is directly preserved from the sorting order proposed by the general-purpose search engine. If we consider for example the question "Generating random words in Java?" described in Section 2, the search proxy supported by Google Web Search returns the relevant posts[3] as listed in Table 1.

3.2 Code Query Generator

The Code Query Generator creates a code search query that augments and structures the free-form query taken by the search proxy (Section 3.1). This augmented query is a list of program elements, such as class and method names (e.g., `Math.random`), as well as natural language terms which can be used to match the documentation.

To generate the augmented query, CoCaBu must extract structural code entities from code snippets embedded in the answers to the questions in the relevant posts returned by the search proxy (Fig. 5b). The code query generator component only considers accepted answers, i.e., answers approved by the Q&A site community.

The augmented query produced by the code query generator is illustrated in Fig. 5c based on the Lucene search engine query format. The reader can observe the following from the illustrated example query whose field semantics are previously described in Table 3:

– terms, excluding stop words, in the user free-form query (i.e., Fig. 5a) are kept, after stemming, in the augmented query (e.g., `code:gener`).

[3] In this illustrative example, we excluded the actual post (http://stackoverflow.com/questions/4951997/generating-random-words-in-java) where this question is asked. To eliminate bias, in all experiments described in Section 5, in which we selected a question of a Q&A site as a subject, we removed the corresponding posts from the list of relevant posts to be used for augmenting the query.

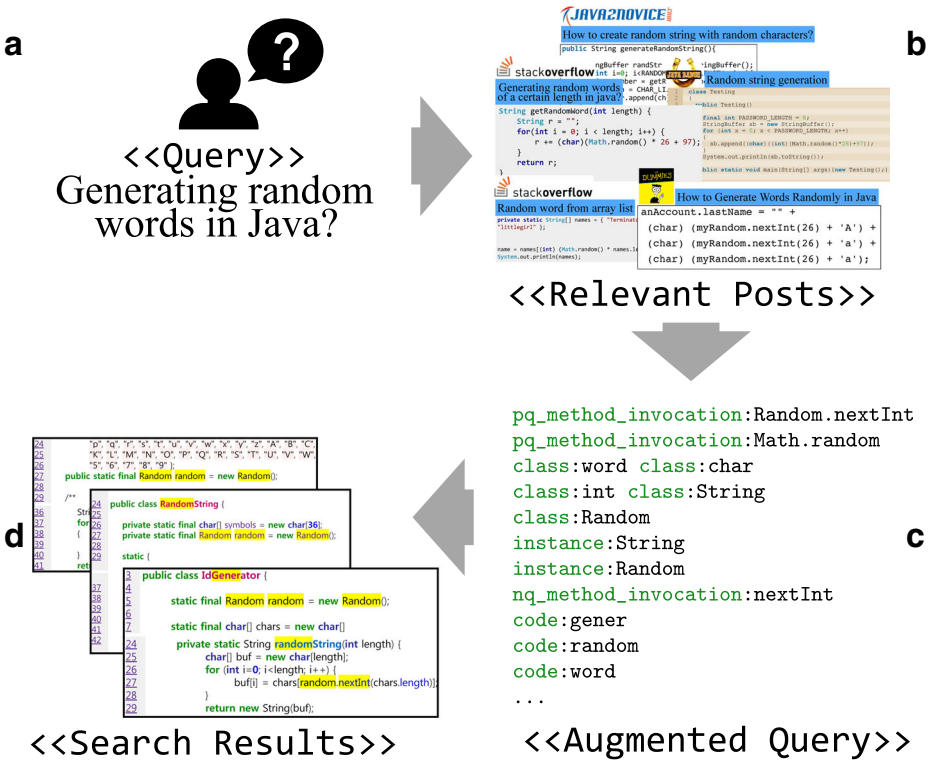


Fig. 5 Illustrative input, intermediate results, and output of a CoCaBu-based code search engine

– structural code entities collected from Q&A snippets (i.e., Fig. 5b) are mentioned with their type (e.g., non-qualified/partially qualified method invocation, or class) in the augmented query (e.g., `pq_method_invocation:Random.nextInt`).

To accelerate code query generation, CoCaBu builds an index of posts. Typically, Q&A forums provide archives of their posts. These posts are often formatted by a structural language such as XML. For example, in `Stack Overflow` posts, code snippets are enclosed in `<code> ...</code>`. As shown in Fig. 6, our approach takes pre-downloaded posts from a Q&A site and extracts metadata (post ID, question title) and code snippets for each post. Each code snippet is then analyzed to retrieve the structural code entities. This phase presents challenges that will be addressed in Section 4.2.

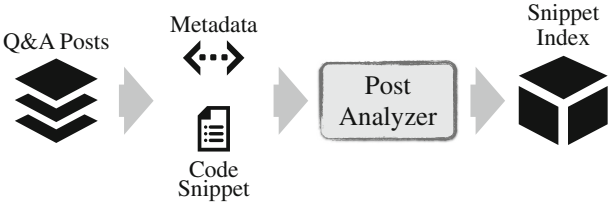


Fig. 6 Creating an index for metadata and code snippets of Q&A posts

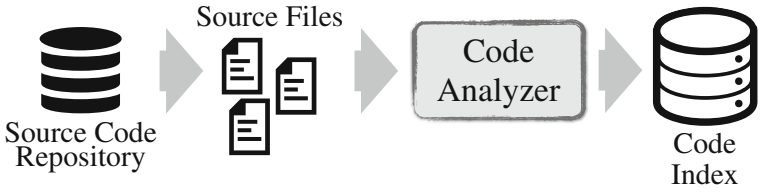


Fig. 7 Creating an index for source code in code repositories up front

Building an index upfront reduces the query generation time when the target post is already indexed. For new posts collected, the component follows the process shown in Fig. 6 to insert it into the index.

3.3 Code Search Engine

The code search engine takes an augmented query from the code query generator and provides a list of search results to the user who issued the original query. The search results are of two granularity levels:

- In case the query is augmented, granularity is further controlled since the structural code entities matched within a source file and the search result can focus on showing only the excerpt with code lines where a match occurred as in the illustrative example of Fig. 3.
- When the query has not been augmented (i.e., the search proxy did not find any Q&A post link within the top ten web search results set), the search engine returns for each result a whole file.

To efficiently provide answers for augmented queries, the code search engine builds an index of source code files found in repositories (cf. Fig. 7). The matching then becomes straightforward as the structural entities in the augmented queries, as well as the NLP terms, are directly search for using the index which will list the most relevant files.

Since the snippet index and the code index (shown in Figs. 6 and 7, respectively) store indices in the same format, full-text search can be effective to obtain search results. Source code files are then the documents while structural code entities represent the search terms.

Once search results are retrieved, the code search engine computes rankings of the source code files based on a scoring function that measures the similarity between the matched files and query terms. The current implementation of CoCaBu uses the scoring function implemented in the Lucene library. This function combines the Boolean Model (BM) and the Vector Space Model (VSM) to determine the relevancy of a document given for a user query[4]. BM is used for reducing the amount of documents that need to be scored by using Boolean logic in the query specification. Each document is represented as a vector $d = (w_1, w_2, ..., w_n)$ where w_i corresponds to the weight of a term occurring in that document. To compute these weights, we use the TF-IDF weighting scheme implemented in Lucene. With these weights, VSM computes the similarity between the documents by using the cosine similarity measure.[5]

[4] https://goo.gl/MqETzP (last accessed 12.07.2015).

[5] https://goo.gl/VPvxnX (last accessed 12.07.2015).

Since displaying the entire content of a source code file is often ineffective for users to understand code examples, the code search engine shows the files after summarizing the content and highlighting lines of code relevant to a given query (Manning et al. 2008). To summarize and highlight search results, CoCaBu uses a query-dependent approach that displays segments of code based on the query terms occurring in the source file. Specifically, the component displays a set of N adjacent lines (the default value is $N = 3$ lines) of code containing the matching query keyword. Finally, we highlight query words occurring in the summarized file to ease their identification.

4 The GitSearch Code Search Engine

This section describes an example instantiation of the CoCaBu approach. We build GitSearch, a code search engine on top of `GitHub` and `Stack Overflow` to explore the large amounts of source code and Q&A posts. In the remainder of this section, we detail the implementation choices that were made in GitSearch.

4.1 Data Collection

To build GitSearch, we selected `Stack Overflow` as the Q&A site where to retrieve relevant developer-approved code snippets. `Stack Overflow` was selected as it is popular among the developer community and it enforces several rules and strategies (e.g., no duplication of question, response voting, marking of accepted answer, rewriting of developer questions for precision and concision, etc.) which make it a fairly representative and reliable dataset of developer questions and answers. For the search proxy, our implementation directly leverages Google web search.[6] User queries are sent to Google Search for retrieving all relevant Q&A posts (i.e., text similarity matching). Note that it is possible for other implementations to use other web search engines including built-in search services of Q&A sites.

We used a dump of `Stack Overflow` posts between July 2008[7] and March 2015 containing 1,363,002 Java and Android tagged questions to build the snippet index. Java was selected in this instantiation since it is one of the most popular programming languages and represents a large developer base (Bissyande et al. 2013a). In this work, we made use of the `posts.xml` documents that have an actual post (i.e., question and answer pair) and other associated metadata such as tags, creation date, question ID, view count of the post, and the score of answers. We then collected posts with snippets in their answers as specified in Section 3.2. In addition, we extracted snippets from answers that were accepted and had a positive score to ensure high quality of code examples. To account for updates in posts, we leveraged the StackExchange REST API[8] with which we could extract metadata and snippets. Users of CoCaBu may collect and use posts from other multiple Q&A forums to extend the opportunity to search for more code snippets.

For the code index, we leverage `GitHub`, the largest repository of open source projects (Kalliamvakou et al. 2014). `GitHub` project data is further widely used by software engineering researchers and practitioners (Bissyande et al. 2013a, 2013b; Thung et al.

[6] www.google.com

[7] We use the dump that contains the oldest data available since the launch of `Stack Overflow` in 2008.

[8] https://api.stackexchange.com/

Table 2 Statistics of collected projects from GitHub

Feature	Value
Number of projects	7,601
Number of files	1,705,677
Number of duplicate files	182,043
Numver of non-Java files	212,680
LOCs	> 297 M

2013). We considered GitHub projects that were forked at least once, to avoid toy and/or inactive projects. Since we focused on Java and Android, we collected GitHub projects in which its major language is "Java" and then removed all non-Java files from the projects when building the code index. As a result, Table 2 shows the statistics of GitHub projects we collected in this work.

4.2 Processing Code Artifacts

Table 3 enumerates structural code entities that GITSEARCH collects when parsing snippets from Q&A posts and source code files from code repositories. These fields are used for indexing documents (source code files and code snippets in Q&A posts) by using Lucene. GITSEARCH leverages these field to make an augmented query as well.

The `import` field contains tokens indexed as import declarations, which can be found at the beginning of Java files (e.g., `java.io.InputStream`). The `super` field represents tokens used as superclass and interface implementation such as type names after `extends` or `implements`. All type names used in a Java file are indexed by the `class` field. For example, `Writer` and `BufferedWriter` are indexed as `used_class` field from this statement `Writer writer = new BufferedWriter(...);`. Names used in methods declarations (e.g., `getValue` in `void getValue(...)`) are indexed in the `method_declaration` field. `nq_method_invocation` and `pq_method_invocation` fields contain indexed tokens from non-qualified and qualified method calls, respectively. For example, in this statement: `obj.nextInt();`, we use `nq_method_invocation` if we cannot resolve type information of `obj`. Otherwise, we use `pq_method_invocation` to make an index of `Random.nextInt` if `obj` is resolved as `Random`. The `instance_creation` field is used for making an index of

Table 3 Structural Code Entities

Field	Description
import	Name of import declarations
super	Direct superclass and implemented interfaces
used_class	Name of used classes
method_declaration	Name of method declarations
nq_method_invocation	Non-qualified method invocations
pq_method_invocation	Partially qualified method invocations
instance_creation	Class instance creations
literal	String Literals
code	tokens in source code after preprocessing

```
URL url = new URL(urlToRssFeed);
SAXParserFactory factory = SAXParserFactory.newInstance();
SAXParser parser = factory.newSAXParser();
XMLReader xmlreader = parser.getXMLReader();
RssHandler theRSSHandler = new RssHandler();
xmlreader.setContentHandler(theRSSHandler);
InputSource is = new InputSource(url.openStream());
xmlreader.parse(is);
return theRSSHandler.getFeed();
```

a Snippet before recovering name qualification.

```
URL url = new URL(urlToRssFeed);
SAXParserFactory factory = SAXParserFactory.newInstance();
SAXParser parser = SAXParserFactory.newSAXParser();
XMLReader xmlreader = SAXParser.getXMLReader();
RssHandler theRSSHandler = new RssHandler();
XMLReader.setContentHandler(theRSSHandler);
InputSource is = new InputSource(URL.openStream());
XMLReader.parse(is);
return RssHandler.getFeed();
```

b Snippet after recovering name qualification.

Fig. 8 Recovery of qualification information

tokens in new object creation such as `Type1` in `Object o = new Type1();`. The `literal` field contains tokens from string constants in a Java file. In addition to other fields, we also use the `code` field to make indices of all tokens in a source code file as plain text after preprocessing. The remainder of this section details our index creation process.

Wrapping code snippets While source code from public repositories is mostly compilable, code snippets from Q&A posts are inherently incomplete since they only include the necessary statements to convey expert responder explanations on a question. Although few code snippets may contain a complete class declaration, in most cases a code snippet consists of a block of code statements. Snippet authors furthermore frequently use ellipses (i.e., "...") before and after code blocks. Thus, GITSEARCH removes ellipses and wraps code snippets by using a custom dummy class and method templates to make it able to parse by standard Java parsers.

Qualifying non-qualified names In addition to wrapping snippets, our approach reasons about qualified names in code snippets. Enclosing class names of methods in snippets are often ambiguous (Dagenais and Robillard 2012) (i.e., method name qualification). For example, Subramanian et al. (2014) found that there are unqualified method name `getId()` more than 27,000 times in their oracle containing 1.6 million types (i.e., classes and method/field signatures) whereas partially qualified name `Node.getId()` can be identified only a few times. This implies that name qualification (even if it is partial) can reduce the size of search space. With a smaller search space, code search tools can more accurately locate and rank the search results. Thus, recovering unqualified names can improve the accuracy of code search.

To recover qualified names of methods, GITSEARCH transforms unqualified names to partially qualified names using structural information collected during AST traversal. Specifically, it converts variable names on which methods are called through their respective classes. Figure 8 illustrates this processing step with an example of code snippet before and after the method qualification.

Text processing In addition to structural entities, our approach collects textual information as well. By treating source code as text, the approach conducts pre-processing such as tokenization (e.g., splitting camel case), stop word removal[9] (Manning et al. 2008), and stemming.

[9]Lucene's (version 4) English default stop word set.

Table 4 Descriptive statistics of the snippet index and code index built from `Stack Overflow` posts and `GitHub` projects, respectively

Feature	Code index from `GitHub`	Snippet index from `Stack Overflow`
# of Documents	1,310,954	298,595
import	8,463,814	86,629
super	1,011,254	35,057
method_declaration	9,056,046	268,929
used_class	23,973,254	1,288,683
nq_method_invocation	771,583	347,917
pq_method_invocation	13,869,622	593,099
instance_creation	2,895,284	220,574
literal	4,079,608	203,762
code	1,101,516	607,070

Indexing With the collected set of information, CoCaBu can build an index of text terms as well as structural code entities found in the source code. To create an index, we build our approach on top of Lucene.[10] Lucene stores data as an index, each consisting of a set of fields, where each field value represents a basic code element for search in our case. Fields are populated with the structural and textual information, produced by the above process, along with the index specific metadata. Further details on this process are provided in Section 3.3.

Table 4 provides a summary of the resulting indices (i.e., the snippet and code indices shown in Figs. 6 and 7) built from `Stack Overflow` posts and `GitHub` open source code repositories.

5 Evaluation

This section describes our evaluation design and reports its results. Our evaluation consists of four studies: a manual verification, online survey, controlled user study, and live study, focusing on answering the following research questions, respectively:

- **RQ1**: Can GitSearch effectively produce relevant code examples for developer queries?
- **RQ2**: Does GitSearch outperform existing code search engines with more acceptable results?
- **RQ3**: Is GitSearch competitive against general search engine for helping to solve programming tasks?
- **RQ4**: Can `Stack Overflow` users accept the search results of GitSearch as answers?

5.1 RQ1: Verification Against a Community Ground Truth

First, we investigate the relevance of the results yielded by GitSearch. To evaluate the relevance, we consider comparing the output code examples against the ground truth of

[10] http://lucene.apache.org

code snippets in answers accepted by the `Stack Overflow` community. This type of verification, which is commonly used in the literature (Bajracharya et al. 2006; Lv et al. 2015), is essential since developers can be quickly deterred by search engine producing many irrelevant results.

Study Design We collect well-known developer questions from `Stack Overflow` posts based on two requirements: (i) a question in a post must relate to "Java" and (ii) its answer must include code snippets. We select the top 10 posts (Q1 – Q10) with the highest 'view count' values (for their questions) to ensure that the study focuses on representative and popular developer tasks. In addition, we randomly collect another 10 more questions (Q11 – Q20) out of top 5,000 `Stack Overflow` posts to avoid the bias of popularity. Table 5 lists the queries used in this study. Note that this process does not bias in favor of our approach. Indeed, for the fair comparison, the actual post where the question is asked is filtered out from the relevant posts, returned by the search proxy that GITSEARCH uses to augment user queries.

We first execute GITSEARCH by giving the title of each question listed in Table 5 as a user query to obtain search results. Since the title often has concise and precise representations of user queries, we focus on the titles instead of descriptions of the posts. Then, we evaluate the top 5 code search examples by GITSEARCH. To assess the relevancy of a GITSEARCH code example, two authors of this paper compared it against the accepted answer on `Stack Overflow` for the associated query. We consider that the example is indeed relevant when it includes the necessary API methods and classes required in the `Stack`

Table 5 Free-form queries used for RQ1 and RQ2

ID	Query terms
Q1	How to add an image to a JPanel?
Q2	How to generate a random alpha-numeric string?
Q3	How to save the activity state in Android?
Q4	How do I invoke a Java method when given the method name as a string?
Q5	Remove HTML tags from a String
Q6	How to get the path of a running JAR file?
Q7	Getting a File's MD5 Checksum in Java
Q8	Loading a properties file from Java package
Q9	How can I play sound in Java?
Q10	What is the best way to SFTP a file from a server?
Q11	Using java.net.URLConnection to fire and handle HTTP requests
Q12	How do I create a file and write to it in Java?
Q13	How do servlets work? Instantiation, sessions, shared variables and multithreading
Q14	Get current stack trace in Java
Q15	Difference between HashMap, LinkedHashMap and TreeMap
Q16	How to convert a char to a String?
Q17	How do I convert a String to an InputStream in Java?
Q18	When do you use Java's @Override annotation and why?
Q19	How to append text to an existing file in Java?
Q20	Java URL encoding of query string parameters

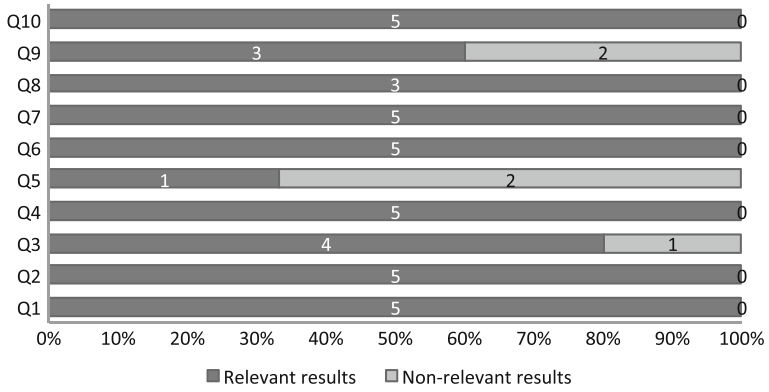


Fig. 9 Relevance of top 5 GITSEARCH results for popular queries (Q1 – Q10) listed in Table 5

`Overflow` answer's code snippet. To increase confidence, both authors must unanimously agree on the relevance of a GITSEARCH result.

Results Figs. 9 and 10 shows that GITSEARCH results are largely relevant to the user query, indirectly demonstrating the accuracy of the query expansion approach.

We also evaluate the effectiveness of GITSEARCH using the $Precision@k$ metric:

$$Precision@k = \frac{1}{|Q|}\sum_{i=1}^{|Q|}\frac{|relevant_{i,k}|}{k} \tag{1}$$

where $relevant_{i,k}$ represents the relevant code search results for query i in the top k returned results, and Q is a set of queries. $Precision@k$ takes an average of all queries whose relevant answers could be found by inspecting the top k (k = 1, 2, 5) of the returned code examples. An effective code search engine should allow developers to find the relevant code examples by examining fewer returned results. Thus, the higher $Precision@k$, the better code search performance. We found that GITSEARCH achieves 90%, 90% and 88%

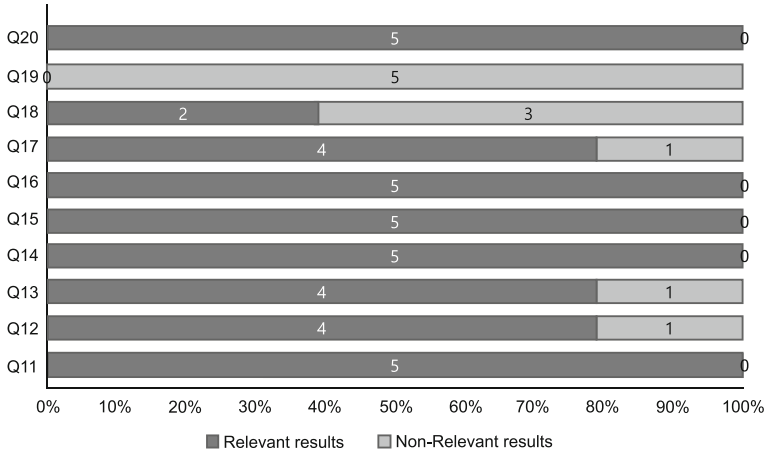


Fig. 10 Relevance of top 5 GITSEARCH results for random queries (Q11 – Q20) listed in Table 5

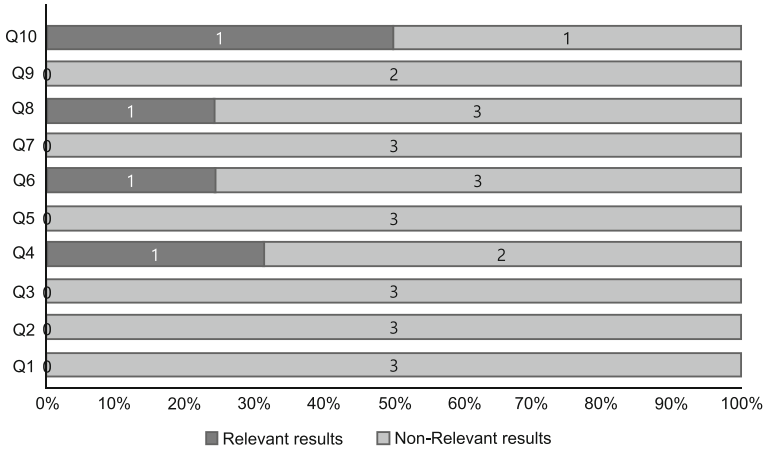


Fig. 11 Relevance of top 5 GITSEARCH results, without query expansion for popular queries (Q1 – Q10) listed in Table 5

scores for $Precision@1$, $Precision@2$, and $Precision@5$, respectively, when applying to the top questions (Q1 – Q10). In addition, for randomly selected questions (Q11 – Q20), $Precision@1$, $Precision@2$, and $Precision@5$ are 80%, 80%, and 78%, respectively. We could not define $recall@k$ because it is impossible to compile the "complete" set of all possible correct answers for a code search query.

To verify the effectiveness of query expansion in GITSEARCH, we conducted another experiment in which our tool has been applied without query expansion. For the same 20 questions in Table 5, the results of code search without query expansion are shown in Figs. 11 and 12. GITSEARCH without query expansion results in $Precision@1$, $Precision@2$, and $Precision@5$ as 20%, 22.5%, and 18.6% for all questions (Q1 – Q20). The precision scores are 20%, 15%, and 13.3% for the top 10 questions (Q1 – Q10), while

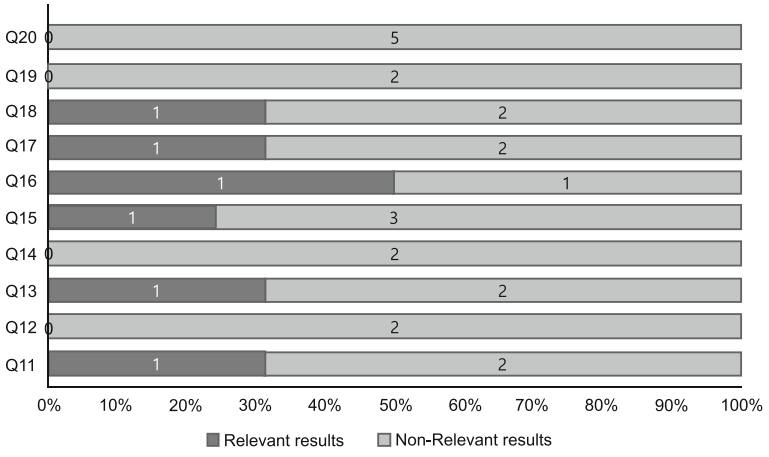


Fig. 12 Relevance of top 5 GITSEARCH results, without query expansion for random queries (Q11– Q20) listed in Table 5

the scores are 20%, 30%, and 24.1% for the random 10 questions (Q11 – Q20). This implies that query expansion in GITSEARCH significantly improves the search quality.

5.2 RQ2: Comparison Against other Code Search Engines

First, we applied the same queries for assessing GITSEARCH (cf. Table 5) to other code search engines: Openhub (2016) and Codota (2016). These code search engines were selected since they are state-of-the-art Internet-scale code search engines and currently available online. On the other hand, we could not compare GITSEARCH against other recent state-of-the-art approaches from the literature because of the reasons listed in Table 6.

The $Precision@k$ values of those two search engines are lower than that of GITSEARCH. OpenHub resulted in 60%, 60%, and 38% scores for $Precision@1$, $Precision@2$, and $Precision@5$, respectively. For Codota, these values are 10%, 10%, and 12%, respectively.

Second, we conduct a user study where we ask developers to check the effectiveness of different code search engines, to assess the usefulness of GITSEARCH from the perspective of practitioners.

Study Design For this study, we recruited participants by posting online survey invitations in software developer communities (750 GitHub, Mozilla, and Eclipse developers, and developers in a Korean company). In all survey invitations, we clearly stated that only developers/students who have Java experience are invited. To facilitate the study, we built a web-based survey tool displaying the code search results from OpenHub, Codota, and GITSEARCH in three anonymized columns. To avoid the bias of people toying with the tool, we only consider the entries of participants who entirely completed the study using the queries in Table 5.

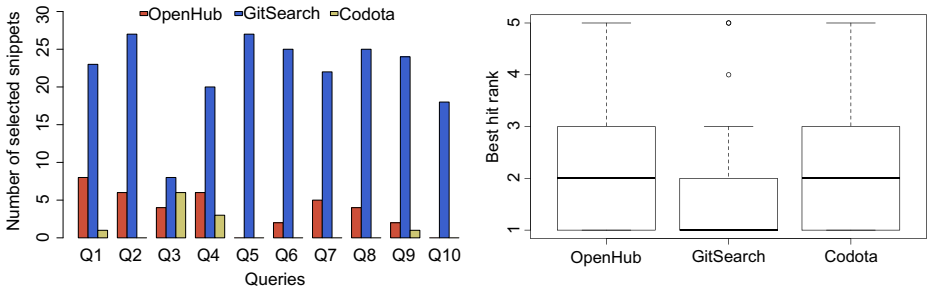
Participants can select code examples based on their preference (i.e., they select their favorite "answers" as it is done with the voting mechanism of `Stack Overflow`). They can select several favorite search results (up to three). The idea, however, being to select only the relevant results, we clearly ask participants to select no result if none is satisfying them. In addition to anonymization, the survey tool excludes the source `Stack Overflow` posts listed in Table 5 from the training data of GITSEARCH to avoid any bias.

Results At the end of the study, we had 47 participants who tried the tool (at least one response). Some of them did not complete the study. Among them, 14 participants completed this study. We randomly sampled 10 answers selected by the participants and manually verified that they were indeed appropriate for resolving the query indicated.

Figure 13a shows the number of selected search results for each code search engine. Participants selected more code examples returned by GITSEARCH than other engines for

Table 6 Unavailability of code search tools and techniques

Portfolio (McMillan et al. 2011) is not available (anymore) and supports only C++.
Exemplar (McMillan et al. 2012) is no longer available.
Sourcerer (Bajracharya et al. 2006)'s team did not reply about the use of their SAS code search engine.
Muse (Moreno et al. 2015) is not relevant: - focuses on API - cannot be queried for snippets.
SNIFF (Chatterjee et al. 2009) engine could not work (issue with the Eclipse plugin).
Keivanloo et al.'s (Keivanloo et al. 2014)' tool is no longer available (lead developer left the project).
CodeHow (Lv et al. 2015) is not available - only a demo video online.



a The number of selected code search results for each search engine.

b Distribution of rankings for the selected search results (the lower the better).

Fig. 13 Comparison between GITSEARCH, Codota and OpenHub

all queries. In particular, the number of selected results was more than double compared to others except for Query Q3.

In addition, we computed the distribution of rankings for the selected search results. If multiple search results of an engine were selected by a user, we counted the highest ranked result only. As shown in Fig. 13b, the median value of GITSEARCH is equal to 1 while the values of other engines are 2.

Discussion Although we could not compare against the most recent CodeHow tool (Lv et al. 2015), note that its authors reported that it produces about 20% more relevant results than OpenHub[11] while Fig. 13a indicates that GITSEARCH provides 50% more relevant results[12] than OpenHub.

5.3 RQ3: Comparison Against General Search Engines

We conducted a comparative study between GITSEARCH and general web search engines. Since many developers rely on general search engines to find solutions to programming tasks, we evaluate the competitiveness of GITSEARCH in comparison to such engines.

Study Design For this study, we recruited 20 graduate students from three universities (Pierre and Marie Curie University in France, University of Luxembourg, and Zhejiang University in China). No author of this paper took part in the study. Each student was asked to find code examples for solving the following two programming tasks from a previous code search study (Lv et al. 2015):

- *Task 1:* Sending emails - write a Java program to read a list of email addresses from a text file, and then send an email with an attachment file to all the email addresses.
- *Task 2:* Image format conversion - write a Java program to read an image in JPEG format, rotate it 180, and then convert it to PNG format.

Participants to the controlled study have been asked to solve one task with GITSEARCH and the other task a general search engine (Google for participants in Europe and Baidu for

[11] Ohloh is now OpenHub.

[12] Despite different queries, our query sets are similar to those of Lv et al. (2015) and representatives of common developer search queries.

participants in China). We consider in this experiment that Google and Baidu are equivalent, and thus we report Google and Baidu results together; indeed, the purpose of the evaluation scenario was not to implicitly compare Google and Baidu. We specify the combinations (task, tool) for every participant by ourselves in order to ensure an even distribution. The tasks were assigned so that every combination (task, tool) is assigned to at least one participant from each university, and all combinations are evenly distributed across the participant pool. Each participant fills a form indicating the different free-form queries used for code search as well as the rank of the returned results that he/she found relevant for the task. We specified that only top 10 results returned by the tools could be examined.

We assess the efficiency of the engines through the Mean Reciprocal Rank (MRR), a statistical metric used to evaluate a process that produces a list of possible responses to a query (Grechanik et al. 2010). The reciprocal rank of a query is the multiplicative inverse of the rank of the first relevant answer. The mean reciprocal rank is the average of the reciprocal ranks of results of a set of queries Q. MRR is computed by using the formula:

$$MRR = \frac{1}{|Q|}\sum_{i=1}^{|Q|}\frac{1}{rank_i} \tag{2}$$

where $rank_i$ represents the rank of the first search results that users find satisfying for query i. MRR values range between 0 and 1, and the higher MRR value the better the performance.

Results Participants to the study entered 77 (37 for Task 1 and 40 for Task 2) distinct free-form queries.

Table 7 shows the percentage of relevant search results that participants in the study marked for the different search engines. GITSEARCH provides more satisfying results for Task 1 while users found more satisfying results with Google/Baidu for Task 2. In contrast, for Task 2, GITSEARCH outperforms Google/Baidu in terms of MRR, returning in higher ranks the satisfying results. On the other hand, GITSEARCH has a lower MRR for Task 1 results. These results suggest that GITSEARCH is competitive against web search engines. We do not, however, take into account the effort required in web search to follow link redirections and parse web pages to find potentially incomplete code snippets. GITSEARCH, on the other hand, provides immediately real-world working code examples.

Discussion We investigated the 77 queries entered by participants in this study. We figured out an interesting pattern: queries entered on web search engines appeared to be more "complete" and more redundant across participants than queries entered on GITSEARCH. Participants to the study admitted that they followed auto-completion suggestions by web search engines.

Table 7 Performance of GITSEARCH vs. general search engine

	Percentage of successful queries†		MRR	
	GITSEARCH	Google/Baidu	GITSEARCH	Google/Baidu
Task 1	**93.76%**	90.00%	0.83	**0.96**
Task 2	75.00%	**100.00%**	**0.89**	0.84

†We compute the ratio of queries having produced satisfying results vs. the total number of queries entered per task

We perform a cross-validation experiment by randomly sampling 10 queries entered by participants on web search engines and use them on GITSEARCH. Similarly, we randomly sample 10 queries entered by participants on GITSEARCH and use them on Google search engine. We record improved MRR values of 0.94 and 0.90 with Task 1 and Task 2 respectively for GITSEARCH. In contrast, MRR values for the web search engine has decreased to 0.72 and 0.65 with Task 1 and Task 2 respectively.

These results suggest a future work on GITSEARCH where we must include logging and feedback mechanisms to record successful queries and propose them to auto-complete queries of future requesters.

5.4 RQ4: Live Study into the Wild

To assess the usefulness of code search engines in Q&A forums, we posted code search results as answers to `Stack Overflow` questions. This study investigates how developers interpret working code examples when they have programming issues. Although GITSEARCH is not designed to directly answer developers' questions, it might help them find a starting point of a programming task. In particular, GITSEARCH can be a good first responder in the context of `Stack Overflow` since there are many unanswered questions (not only "no answer selected by questioners" but also literally "no answer") in `Stack Overflow`.

Study Design We monitored questions with **Java** tags and selected 25 out of them based on the following criteria:

- Questions about Java programming.
- "How-To" questions such as "List all files in resources directory in java project".
- No tool usage questions such as "How to create a project in Eclipse?"
- No conceptual questions such as "What is the difference between **A** or **B**" and "why this class is so slow?".
- Questions not answered by anyone yet.

For each question, we extracted its title and put it into GITSEARCH to obtain code search results. We took the topmost result among the search results and posted it as an answer. The answer consists of 1) the most relevant code fragments selected by GITSEARCH and 2) hyperlink for the original source code where GITSEARCH found the fragments from. The latter is important since developers can figure out more context about the working code examples. In addition, we repeated the same procedure with OpenHub to compare its effectiveness with our technique. We do not post Codota's results on `Stack Overflow` to avoid "spamming" requesters, as we could see ourselves that its topmost result was irrelevant for most questions.

Results GITSEARCH could answer more questions than OpenHub as shown in Table 8 ("Resp" of "#Ans"). GITSEARCH responded 25 questions by using its search results while OpenHub did only for 18 out of 25 questions. For the other seven questions, OpenHub could not produce any search result for reasons that are unknown to us (perhaps, due to an issue of the engine's query matching implementation). In addition, at the end of the study, `Stack Overflow` users eventually answered only 8 out of 25 questions. Note that three answers were accepted by the questioners among the 25 answers by GITSEARCH. None of 18 answers by OpenHub were accepted. For human answers, questioners accepted four out of 8 answers.

Table 8 Results of our live study between GITSEARCH, OpenHub, and Human.

	#Ans		Upvotes		Downvotes		Pos. Comm.		Neg. Comm.		Most
	Resp	Acc	$\|x\|$	Σ	$\|x\|$	Σ	$\|x\|$	Σ	$\|x\|$	Σ	voted?
GITSEARCH	25	3	6	7 (0.28)	5	10 (0.40)	7	7 (0.28)	3	5 (0.20)	4 (0.16)
OpenHub	18	0	2	3 (0.17)	2	3 (0.17)	2	2 (0.11)	2	3 (.17)	0
Human	8	4	4	9 (1.13)	1	1 (0.13)	3	6 (0.75)	2	2 (0.25)	3 (0.38)

"Resp" in "#Ans" is the number of questions answered by each technique while "Acc" is the number of answers accepted by the questioners. "Up and down" votes are the number of votes given by `Stack Overflow` users. "Pos." and "Neg. Comm." comments are positive and negative comments made by the users for each answer. "Most voted?" represents the number of answers that received the most number of upvotes. $|x|$ indicates the number of answers with at least one up/down vote and positive/negative comment. Σ is the sum of occurrences while the numbers in parentheses are average (i.e., $\Sigma/Resp$)

Our technique received more up and downvotes than OpenHub while human answers took more upvotes and less downvotes. Six out of 25 answers by GITSEARCH received at least one upvote while other five of them took at least one downvote (six upvotes and 10 downvotes in total). Four of the six were the most-upvoted answers in their posts. OpenHub's answers had only two upvotes and three downvotes, respectively. Human answers took 9 upvotes (from four different answers) and 1 downvote (note that a single answer took 4 upvotes) where three answers were most-voted. In `Stack Overflow`, votes imply that those users would encourage (or discourage) the answer. While its up and downvotes were almost tied with human results, it is obvious that GITSEARCH had more interest from users than OpenHub.

In addition, GITSEARCH initiated user discussions more frequently. We counted comments made by `Stack Overflow` users and examined whether each comment is positive and negative. Our answers took 7 positive and 5 negative comments while OpenHub's results were followed by two and three, respectively. GITSEARCH does not explicitly outperform human answers (6 positive and 2 negative) but note that there were 17 of out 25 questions unanswered yet by human users. For the 17 questions, our technique answered them and received one upvote and three downvotes as well as three positive and two negative comments.

Discussion The results of this study implies that GITSEARCH can be a better first responder than OpenHub. As shown in Table 8, many questions in `Stack Overflow` remain unanswered for several days. Our technique can provide a starting point for questions even if they are not complete answers as many users would follow up the answers by giving their votes and adding comments. Once users are interested in a question, there might be more probability to discuss solutions for the question.

In addition, code search results by GITSEARCH can be selected by `Stack Overflow` users as accepted answers, which implies that the results are highly relevant and appropriate to the questions. For three out of 25 questions, the questioners accepted our results even though the answers have only code excerpt from real source code without any additional explanation. Note that a questioner can select only one answer as the accepted one. This may indicate that questioners would take advantage of code search results to deal with their problems shown in the question. Furthermore, this can imply that code search engines would be an automatic answer generator for some questions in `Stack Overflow` if their accuracy is improved.

5.5 Threats to Validity

The design of CoCaBu and the implementation of GitSearch raises a number of threats to validity that we have tried to mitigate. We list them below:

Internal validity the user study was performed with a limited total number of 34 (=14+20) participants compared to the large number of participants used by Muse (Moreno et al. 2015) authors for their API example search engine. However, among free-form code search works, some do not perform user studies (e.g., Bajracharya et al. 2006), while others use fewer participants than us (e.g., CodeHow (20), Portfolio (19), SNIFF (undisclosed)). We have attempted to reach representativity by inviting professional developers as well as graduate students.

In addition, throughout the live study (Section 5.4), we tried to take feedback from `Stack Overflow` overflow users in the loop of problem-solving. This implies that an additional number of participants were involved in our evaluation.

External validity we used only English as a query language, focused on Java-related questions, and explored only `Stack Overflow` and `GitHub` in our implementation. This threat should be limited by the fact that (1) English is a popular language in the programming community, (2) Java is one of the most popular programming languages, and furthermore, (3) `GitHub` and `Stack Overflow` are the largest code hosting site and Q&A forum respectively.

Construct validity we only focus on queries without the exact name of APIs. This threat, however, is limited since for new tasks, developers often do not know the name of the relevant APIs (Hoffmann et al. 2007).

6 Related Work

There are several lines of research work that relate to our approach. We list their main contributions in each category.

6.1 API Usage Examples Search

Recently, there have been a number of code search techniques (Moreno et al. 2015; Bajracharya et al. 2010; Keivanloo et al. 2014; Mandelin et al. 2005; Thummalapenta and Xie 2007; Gu et al. 2016), focusing on locating API usage examples. Searching for specific API usages is a subset of code search activities. Compared to general code search, developers tend to be aware of the exact (or similar) name of a target API, which facilitates the search. Thus, these techniques focus on creating an index of API call sites only.

Moreno et al. (2015) proposed Muse, an approach to mining and ranking code examples that show how to use a given method. Muse and CoCaBu differ on three main aspects. First, CoCaBu supports free-form queries, while Muse takes as input an API method signature. Second, Muse provides a code snippet for a specific method. CoCaBu, on the other hand, is not attached to a single API, and shows a set if APIs used to solve the task at hand. Lastly, Muse requires fully compilable client projects in order to apply static slicing. Note that CoCaBu is able to handle incomplete source code.

Chatterjee et al. presented SNIFF (Chatterjee et al. 2009), a technique that combines API documentation with publicly available Java code. SNIFF annotates each method call statement with its corresponding API documentation. This allows free-form English queries about the task at hand, which relaxes the need to know the appropriate API beforehand. Although SNIFF returns usage code examples as well, it requires a fully compilable code unit and the accompanying API documentation as well as external libraries. Additionally, the before-mentioned code intersection is not suitable for Internet-scale code search, because it has a complexity of $O(n^2)$, where n is the number of hits.

6.2 Source Code Search

There have been several approaches to code search, which are relevant to CoCaBu. CodeHow, Sourcerer and Portfolio constitute the state-of-the-art of such approaches in the literature. CodeHow (Lv et al. 2015) leverages code documentation to recognize the potential APIs a query refers to and expands the query with these APIs to improve the accuracy of the search results. In contrast, CoCaBu assumes that 1) documentation is not always available, and 2) leveraging independent API documentation may create noise in a query whose answer requires a specific set of related APIs. Furthermore, CoCaBu augments queries based on information of code terms in source code snippets.

Sourcerer (Bajracharya et al. 2006) is an infrastructure that facilitates the collection and analysis of large-scale open-source repositories. On top of that infrastructure, Sourcerer provides programmatic access to all the artifacts stored and managed through a set of services. Sourcerer crawls Java projects from several types of code repositories such open code repositories (e.g. Sourceforge and Apache) and web sites. Similar to CoCaBu, Sourcerer leverage structural code information to perform fine-grained code search. However, the construction of the search index requires a complete compilation unit (i.e., all dependencies must be resolved). Moreover, we exploit high-quality code snippets from `Stack Overflow` to improve the quality of code search results.

Portfolio (McMillan et al. 2011) retrieves and visualizes relevant functions and their usage scenarios to highlight a chain of function invocations. To realizes their objective, Portfolio computes the textual similarity between a user query and the function signatures. Subsequently, a function call graph is employed to locate functions which are relevant to a task, even if those function signatures do not include any keywords of the query. Compared to Portfolio, CoCaBu focuses on usage examples that answer complex queries by leveraging `Stack Overflow` code snippets.

OpenHub Code Search (Openhub 2016) (formerly ohloh.net) is a free web-based code search engine. Although OpenHub has indices of more than 21 billion lines of code collected from open source projects on the Internet, it directly matches query terms with terms in source files. This is a common limitation of several Internet-scale search engines, including Codota (2016). Contrary to them, we resolve the vocabulary mismatch problem by augmenting user queries.

6.3 Query Reformulation

The literature of software engineering research in general, and code search in particular, includes a number of approaches dealing with query reformulations. Haiduc et al. (2013) proposed a query reformulation strategy leveraging machine learning on a set of historical queries and associated relevant results. In contrast, CoCaBu does not require labelled data (which can be expensive to build for a large scale engine). Exemplar (Grechanik et al.

2010) uses help documents to expand queries while CoCaBu accounts for the fact that a number of relevant code examples in repositories are actually not documented. Finally, CodeExchange (Martie et al. 2015) further refines textual methods by exploiting relationships between successive user queries. Such an approach can complement CoCaBu-based implementations of search engines by further improving the selection of expansion tokens.

There are broadly two ways of reformulating a query: a global approach would use a thesaurus, like WordNet, to enumerated related words and synonyms of the query terms; a more local approach, however, iteratively tries to expand the query by considering extra terms appearing in initial results obtained with the original query and which are marked as relevant by the searcher. Query expansion has been shown to be effective in many natural language processing (NLP) tasks (Carpineto et al. 2001; Xu and Croft 1996). In code search research, query expansion has been intensively used in recent years: Wang et al. (2014) consider human intervention to rank search results. Sisman and Kak (2013) also proposed to leverage user feedback for searching code in the context of bug localization. CONQUER (Shepherd et al. 2007; Roldan-vega et al. 2013; Hill et al. 2014) refines the queries by suggesting the most highly co-occurring words that appear in the source code as alternative words. More recently, Lu et al. (2015) developed a query reformulation technique based on part-of-speech of each word in queries and WordNet. Lemos et al. (2014) proposed AQE, an automatic query expansion approach, which uses test cases as inputs, and leverages WordNet and SwordNet (Yang and Tan 2014), a code-related thesaurus, to expand queries.

To deal with the search on structured data in the web (e.g., databases of IT consumables), Gollapudi et al. (2011) have proposed to rewrite web search queries expanding them to include tags on the nature of each token (e.g., brand, display size, etc.), thus creating a structured query. Likewise, CoCaBu implements a query structuration approach, to improve accuracy. Closely related to our work, QECK (Nie et al. 2016), concurrently developed with CoCaBu, automatically extracts software-specific expansion words from Q&A posts on Stack Overflow: the idea is to identify what are the most recurrent code terms that are often associated with specific query terms. Our approach, however, is more refined as we focus on Q&A posts which are mostly related to the user input query.

6.4 Miscellaneous

Code recommendation Recommendation engines assist developers in their use of complex libraries or frameworks by presenting them with reusable code fragments in other locations of their code, with documentation, or with pointers to blogs and Q&A sites. Strathcona (Holmes and Murphy 2005) is an approach in which a query is generated from a user's source code and matched with an example repository that uses a target library of the framework. They thus require prior knowledge of the relevant library.

Prompter (Ponzanelli et al. 2014), on the contrary, does not provide code snippets but matches the current code context with relevant `Stack Overflow` posts. The technique relies on different features to capture the similarity between `Stack Overflow` discussions and the current code context. On the other hand, our approach does not recommend discussions but use `Stack Overflow`'s code snippets to search for similar usage examples in a large code repository.

Stack Overflow Several studies have explored `Stack Overflow` questions and answers (Nasehi et al. 2012; Barzilay et al. 2013; Mamykina et al. 2011; Treude and Robillard 2016). However, to the best of our knowledge, its data has never been leveraged to improve code search engine results.

7 Conclusion

We have presented CoCaBu, a novel approach to addressing the vocabulary mismatch problem in code search. CoCaBu augments free-form queries by leveraging code snippets in answers of related posts from Q&A sites. The key insight from our work is that it is possible to map human concepts expressed in queries (which are often written with similar terms by developers) with structural code entities (which are the most relevant terms for matching source code with high relevance). We implemented a code search engine, GitSearch, following the CoCaBu approach for the `GitHub` super-repository of projects. To that end, we leveraged `Stack Overflow` posts to find the best mappings between developer query terms and structural code entities. Our evaluation with user studies demonstrated that GitSearch outperforms Internet-scale code search engines and is competitive against established web search engines for resolving programming tasks. We also found with a live study that users in Q&A forums show interest in the real-world code examples yielded by GitSearch.

Acknowledgments The authors would like to thank the anonymous reviewers for their helpful comments and suggestions. This work was supported by the Fonds National de la Recherche (FNR), Luxembourg, under projects RECOMMEND C15/IS/10449467, FIXPATTERN C15/IS/9964569, FNR-AFR PhD/11623818, and by the Singapore Ministry of Education (MOE) Academic Research Fund (AcRF) Tier 1 grant, under project 16-C220-SMU-004.

References

Bajracharya SK, Ngo T, Linstead E, Dou Y, Rigor P, Baldi P, Lopes CV (2006) Sourcerer: a search engine for open source code supporting structure-based search. In: Proceedings of the companion to the 21st ACM SIGPLAN symposium on object-oriented programming systems, languages, and applications (OPSLA). Portland, Oregon, USA, pp 681–682

Bajracharya SK (2010) Facilitating internet-scale code retrieval. Ph.D. thesis, Long Beach. AAI3422111

Bajracharya SK, Ossher J, Lopes CV (2010) Leveraging usage similarity for effective retrieval of examples in code repositories. In: Proceedings of the 18th ACM SIGSOFT international symposium on foundations of software engineering (FSE). Santa Fe, New Mexico, USA, pp 157–166

Barzilay O, Treude C, Zagalsky A (2013) Facilitating crowd sourced software engineering via stack overflow. In: Finding source code on the web for remix and reuse. Springer, Berlin, pp 289–308

Bissyande T, Thung F, Lo D, Jiang L, Reveillere L (2013) Popularity, interoperability, and impact of programming languages in 100,000 open source projects. In: Computer software and applications conference (COMPSAC), 2013 IEEE 37th annual, pp 303–312. https://doi.org/10.1109/COMPSAC.2013.55

Bissyandé TF, Thung F, Lo D, Jiang L, Réveillère L (2013) Orion: a software project search engine with integrated diverse software artifacts. In: ICECSS

Carpineto C, de Mori R, Romano G, Bigi B (2001) An information-theoretic approach to automatic query expansion. ACM Trans Inf Syst 19(1):1–27. https://doi.org/10.1145/366836.366860

Chatterjee S, Juvekar S, Sen K (2009) Sniff: a search engine for java using free-form queries. In: Fundamental approaches to software engineering. Springer, Berlin, pp 385–400

Chen TH, Thomas SW, Nagappan M, Hassan AE (2012) Explaining software defects using topic models. In: Proceedings of the 9th IEEE working conference on mining software repositories, MSR '12. IEEE Press, Piscataway, pp 189–198. http://dl.acm.org/citation.cfm?id=2664446.2664476

Cleland-Huang J, Czauderna A, Gibiec M, Emenecker J (2010) A machine learning approach for tracing regulatory codes to product specific requirements. In: ACM/IEEE 32Nd international conference on software engineering, vol 1, pp 155–164. https://doi.org/10.1145/1806799.1806825

Codota (2016) http://www.codota.com. Last accessed 12.03.2016

Dagenais B, Robillard MP (2012) Recovering traceability links between an API and its learning resources. In: Proceedings of the 34th international conference on software engineering (ICSE). IEEE, Piscataway, pp 47–57

Eckert K, Stuckenschmidt H, Pfeffer M (2007) Interactive thesaurus assessment for automatic document annotation. In: Proceedings of the 4th international conference on knowledge capture, k-CAP '07. ACM, New York, pp 103–110. https://doi.org/10.1145/1298406.1298426

Furnas GW, Landauer TK, Gomez LM, Dumais ST (1987) The vocabulary problem in human-system communication. Commun ACM 30(11):964–971. https://doi.org/10.1145/32206.32212
Gallardo-Valencia RE, Elliott Sim S (2009) Internet-scale code search. In: Proceedings of the 2009 workshop on search-driven development-users, infrastructure, tools and evaluation, SUITE
Gollapudi S, Ieong S, Ntoulas A, Paparizos S (2011) Efficient query rewrite for structured web queries. In: Proceedings of the 20th ACM international conference on information and knowledge management, CIKM '11. ACM, New York, pp 2417–2420. https://doi.org/10.1145/2063576.2063981
Grechanik M, Fu C, Xie Q, McMillan C, Poshyvanyk D, Cumby C (2010) A search engine for finding highly relevant applications. In: 2010 ACM/IEEE 32nd international conference on software engineering, vol 1, pp 475–484. https://doi.org/10.1145/1806799.1806868
Gu X, Zhang H, Zhang D, Kim S (2016) Deep api learning. In: International symposium on foundations of software engineering (FSE)
Haiduc S, Bavota G, Marcus A, Oliveto R, De Lucia A, Menzies T (2013) Automatic query reformulations for text retrieval in software engineering. In: Proceedings of the 2013 international conference on software engineering. IEEE Press, Piscataway, pp 842–851
Haiduc S, De Rosa G, Bavota G, Oliveto R, De Lucia A, Marcus A (2013) Query quality prediction and reformulation for source code search: The refoqus tool. In: Proceedings of the 2013 international conference on software engineering, ICSE '13. IEEE Press, Piscataway, pp 1307–1310. http://dl.acm.org/citation.cfm?id=2486788.2486991
Hill E, Roldan-vega M, Fails JA, Mallet G (2014) NL-based query refinement and contextualized code search results: a user study. In: 2014 Software evolution week - IEEE conference on software maintenance, reengineering, and reverse engineering, CSMR-WCRE 2014, Antwerp, Belgium, February 3-6, 2014, pp 34–43. https://doi.org/10.1109/CSMR-WCRE.2014.6747190
Hoffmann R, Fogarty J, Weld DS (2007) Assieme: finding and leveraging implicit references in a web search interface for programmers. In: Proceedings of the 20th annual ACM symposium on user interface software and technology (UIST). Newport, Rhode Island, USA, pp 13–22
Holmes R, Murphy GC (2005) Using structural context to recommend source code examples. In: Proceedings of the 27th international conference on software engineering (ICSE). St. Louis, MO, USA, pp 117–125
Kalliamvakou E, Gousios G, Blincoe K, Singer L, German DM, Damian D (2014) The promises and perils of mining GitHub. In: Proceedings of the 11th working conference on mining software repositories (MSR). Hyderabad, India, pp 92–101
Keivanloo I, Rilling J, Zou Y (2014) Spotting working code examples. In: Proceedings of ICSE
Kim S, Kim D (2016) Automatic identifier inconsistency detection using code dictionary. Empir Softw Eng (EMSE) 21(2):565–604
Lemos OAL, de Paula AC, Zanichelli FC, Lopes CV (2014) Thesaurus-based automatic query expansion for interface-driven code search. In: Proceedings of the 11th working conference on mining software repositories (MSR). Hyderabad, India, pp 212–221
Liu LM, Halper M, Geller J, Perl Y (1999) Controlled vocabularies in oodbs: Modeling issues and implementation. Distrib. Parallel Databases 7(1):37–65. https://doi.org/10.1023/A:1008682210559
Lozano A, Kellens A, Mens K (2011) Mendel: Source code recommendation based on a genetic metaphor. In: Proceedings of the 2011 26th IEEE/ACM international conference on automated software engineering, ASE '11. IEEE Computer Society, Washington, pp 384–387. https://doi.org/10.1109/ASE.2011.6100078
Lu M, Sun X, Wang S, Lo D, Duan Y (2015) Query expansion via WordNet for effective code search. In: Proceedings of 22nd IEEE international conference on software analysis, evolution, and reengineering (SANER). Montreal, QC, Canada, pp 545–549
Lv F, Zhang H, guang Lou J, Wang S, Zhang D, Zhao J (2015) Codehow: effective code search based on api understanding and extended boolean model (e). In: 30th IEEE/ACM international conference on automated software engineering (ASE), pp 260–270
Mamykina L, Manoim B, Mittal M, Hripcsak G, Hartmann B (2011) Design lessons from the fastest Q&A site in the west. In: Proceedings of the SIGCHI conference on human factors in computing systems (CHI). Vancouver, BC, Canada, pp 2857–2866
Mandelin D, Xu L, Bodík R, Kimelman D (2005) Jungloid mining: helping to navigate the api jungle. ACM SIGPLAN Not 40(6):48–61
Manning CD, Raghavan P, Schütze H (2008) Introduction to information retrieval. Cambridge University press, New York
Martie L, LaToza TD, van der Hoek A (2015) CodeExchange: supporting reformulation of internet-scale code queries in context (T). In: 2015 30th IEEE/ACM international conference on Automated software engineering (ASE). Lincoln, USA, pp 24–35
McMillan C, Grechanik M, Poshyvanyk D, Fu C, Xie Q (2012) Exemplar: a source code search engine for finding highly relevant applications. IEEE Trans Softw Eng 38(5):1069–1087. https://doi.org/10.1109/TSE.2011.84

McMillan C, Grechanik M, Poshyvanyk D, Xie Q, Fu C (2011) Portfolio: finding relevant functions and their usage. In: Proceedings of ICSE

Moreno L, Bavota G, Di Penta M, Oliveto R, Marcus A (2015) How can i use this method? In: ICSE

Nasehi SM, Sillito J, Maurer F, Burns C (2012) What makes a good code example?: a study of programming Q&A in stackoverflow. In: Proceedings of 28th IEEE international conference on software maintenance (ICSM). Trento, Italy, pp 25–34

Nguyen AT, Nguyen TT, Al-Kofahi J, Nguyen HV, Nguyen T (2011) A topic-based approach for narrowing the search space of buggy files from a bug report. In: 26Th IEEE/ACM international conference on automated software engineering (ASE), pp 263–272. https://doi.org/10.1109/ASE.2011.6100062

Nie L, Jiang H, Ren Z, Sun Z, Li X (2016) Query expansion based on crowd knowledge for code search. IEEE Trans Serv Comput 9(5):771–783. https://doi.org/10.1109/TSC.2016.2560165

Openhub (2016) http://code.openhub.net. Last accessed 12.03.2016

Ponzanelli L, Bavota G, Di Penta M, Oliveto R, Lanza M (2014) Mining stackoverflow to turn the IDE into a self-confident programming prompter. In: Proceedings of the 11th working conference on mining software (MSR). Hyderabad, India, pp 102–111

Roldan-vega M, Mallet G, Hill E, Fails JA (2013) Conquer: a tool for nl-based query refinement and contextualizing source code search results. In: Proceedings 29th IEEE international conference on software maintenance. Citeseer

Ruthven I (2003) Re-examining the potential effectiveness of interactive query expansion. In: Proceedings of the 26th annual international ACM SIGIR conference on research and development in informaion retrieval, SIGIR '03. ACM, New York, pp 213–220. https://doi.org/10.1145/860435.860475

Sadowski C, Stolee KT, Elbaum S (2015) How developers search for code: a case study. In: Proceedings of the 2015 10th joint meeting on foundations of software engineering, ESEC/FSE 2015. ACM, New York, pp 191–201. https://doi.org/10.1145/2786805.2786855

Shepherd D, Fry ZP, Hill E, Pollock L, Vijay-Shanker K (2007) Using natural language program analysis to locate and understand action-oriented concerns. In: Proceedings of the 6th international conference on aspect-oriented software development (AOSD). Vancouver, British Columbia, Canada, pp 212–224

Sisman B, Kak AC (2013) Assisting code search with automatic query reformulation for bug localization. In: Proceedings of the 10th working conference on mining software repositories (MSR). San Francisco, CA, USA, pp 309–318

Stylos J, Myers BA (2006) Mica: a web-search tool for finding API components and examples. In: IEEE symposium on Visual languages and human-centric computing, 2006. VL /HCC 2006, pp 195–202. https://doi.org/10.1109/VLHCC.2006.32

Subramanian S, Inozemtseva L, Holmes R (2014) Live API documentation. In: Proceedings of the 36th international conference on software engineering (ICSE). Hyderabad, India, pp 643–652

Thummalapenta S, Xie T (2007) Parseweb: a programmer assistant for reusing open source code on the web. In: Proceedings of the 22nd IEEE/ACM international conference on automated software engineering (ASE). Atlanta, Georgia, USA, pp 204–213

Thung F, Bissyande TF, Lo D, Jiang L (2013) Network structure of social coding in Github. In: Proceedings of the 17th European conference on Software maintenance and reengineering (CSMR). Genova, Italy, pp 323–326

Treude C, Robillard M (2016) Augmenting api documentation with insights from stack overflow. In: Proceedings of the 38th international conference on software engineering, ICSE '16, pp 392–403

Wang S, Lo D, Jiang L (2014) 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). Vasteras, Sweden, pp 677–682

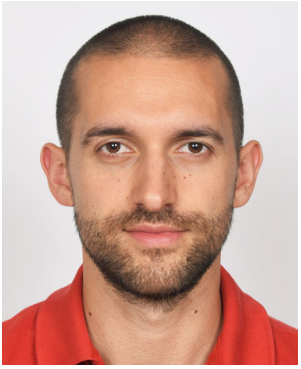
Xie T, Pei J (2006) Mapo: mining api usages from open source repositories. In: Proceedings of the 2006 international workshop on mining software repositories, MSR '06. ACM, New York, pp 54–57. https://doi.org/10.1145/1137983.1137997

Xu J, Croft WB (1996) Query expansion using local and global document analysis. In: Proceedings of the 19th annual international ACM SIGIR conference on research and development in information retrieval (SIGIR). Zurich, Switzerland, pp 4–11

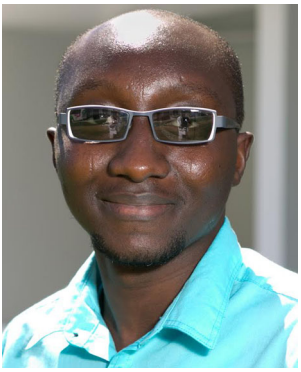
Yang J, Tan L (2014) Swordnet: inferring semantically related words from software context. Empir Softw Eng 19(6):1856–1886

Zhao L, Callan J (2010) Term necessity prediction. In: Proceedings of the 19th ACM international conference on information and knowledge management, CIKM

Zhao L, Callan J (2012) Automatic term mismatch diagnosis for selective query expansion. In: Proceedings of the 35th international ACM SIGIR conference on research and development in information retrieval, SIGIR



Raphael Sirres holds a Master degree in Computer Science from University of Luxembourg since 2015. Currently, he is working at the National Library of Luxembourg, and responsible for the development of discovery and linking services. His subject of interests includes Natural Language Processing, Linked Data and Information retrieval.



Tegawendé F. Bissyandé is a research scientist at the Interdisciplinary Centre for Security, Reliability and Trust (SnT) of the University of Luxembourg. He received his PhD degree in Computer Sciences from the University of Bordeaux (France) in 2013. His research interests lie in Debugging and fixing software and in Empirical studies for improving software engineering processes.

Dongsun Kim received the BEng, MS, and PhD degrees in computer science and engineering from Sogang University, Seoul, Korea, in 2003, 2005, and 2010, respectively. He is currently a research associate at the University of Luxembourg. His research interests include mining software repositories, automatic patch generation, static analysis, search-based software engineering (SBSE).

David Lo received his PhD degree from the School of Computing, National University of Singapore in 2008. He is currently an Associate Professor in the School of Information Systems, Singapore Management University. He has close to 10 years of experience in software engineering and data mining research and has more than 200 publications in these areas. He received the Lee Foundation Fellow for Research Excellence from the Singapore Management University in 2009, and a number of international research awards including several ACM distinguished paper awards for his work on software analytics. He has served as general and program co-chair of several well-known international conferences (e.g., IEEE/ACM International Conference on Automated Software Engineering), and editorial board member of a number of high-quality journals (e.g., Empirical Software Engineering).

Jacques Klein is senior research scientist at the University of Luxembourg, and at the Interdisciplinary Centre for Security, Reliability and Trust (SnT). He received his Ph.D. degree in Computer Science from the University of Rennes, France in 2006. His main areas of expertise are threefold: (1) Mobile Security (malware detection, prevention and dissection, static analysis for security, vulnerability detection, etc.); (2) Software Reliability (software testing, semi-automated and fully-automated program repair, etc.); (3) Data Analytics (multi-objective reasoning and optimization, model-driven data analytics, time series pattern recognition, text mining, etc.). In addition to academic achievements, Dr. Klein has also standing experience and expertise on successfully running industrial projects with several industrial partners in various domains by applying data analytics, software engineering, information retrieval, etc., to their research problems.

Kisub Kim is a PhD student at the University of Luxembourg. He received the B.S.E and M.Eng degrees in computer engineering from Chungbuk National University, Cheongju, Korea, in 2014 and 2017, respectively. He worked in Kyunghee University Medical Center, as a software developer during 2014 and 2015. His research interests include source code search, mining software repositories, and requirement engineering.

Yves Le Traon is professor at University of Luxembourg, in the domain of software engineering, testing, security and model-driven engineering. He received his engineering degree and his PhD in Computer Science at the "Institut National Polytechnique" in Grenoble, France, in 1997. From 1998 to 2004, he was an associate professor at the University of Rennes, in Brittany, France. From 2004 to 2006, he was an expert in Model-Driven Architecture and Validation at "France Te le ćom R&D". In 2006, he became professor at Tele- com Bretagne (Ecole Nationale des Tlcommunications de Bretagne). He is currently the head of the CSC Research Unit (e.g. Department of Computer Science) at University of Luxembourg. He is a member of the Interdisciplinary Centre for Security, Reliability and Trust (SnT), where he leads the research group SERVAL (SEcurity Reasoning and VALidation). His research interests include software testing, model-driven engineering, model based testing, evolutionary algorithms, software security, security policies and Android security. The current key-topics he explores are related to Internet of things (IoT), Big Data (stress testing, multi-objective optimization and data protection), and mobile security and reliability. He is author of more than 140 publications in international peer-reviewed conferences and journals.