FaCoY – A Code-to-Code Search Engine

Kisub Kim[1], Dongsun Kim[1*], Tegawendé F. Bissyandé[1*],
Eunjong Choi[2], Li Li[3*], Jacques Klein[1], Yves Le Traon[1]
[1]SnT, University of Luxembourg - Luxembourg
[2]Nara Institute of Science and Technology (NAIST) - Japan
[3]Faculty of Information Technology, Monash University - Australia

ABSTRACT

Code search is an unavoidable activity in software development. Various approaches and techniques have been explored in the literature to support code search tasks. Most of these approaches focus on serving user queries provided as natural language free-form input. However, there exists a wide range of use-case scenarios where a code-to-code approach would be most beneficial. For example, research directions in code transplantation, code diversity, patch recommendation can leverage a code-to-code search engine to find essential ingredients for their techniques. In this paper, we propose FaCoY, a novel approach for statically finding code fragments which may be *semantically* similar to user input code. FaCoY implements a *query alternation* strategy: instead of directly matching code query tokens with code in the search space, FaCoY first attempts to identify other tokens which may also be relevant in implementing the functional behavior of the input code. With various experiments, we show that (1) FaCoY is more effective than online code-to-code search engines; (2) FaCoY can detect more semantic code clones (i.e., Type-4) in BigCloneBench than the state-of-the-art; (3) FaCoY, while static, can detect code fragments which are indeed similar with respect to runtime execution behavior; and (4) FaCoY can be useful in code/patch recommendation.

1 INTRODUCTION

In software development activities, source code examples are critical for understanding concepts, applying fixes, improving performance, and extending software functionalities [6, 46, 63, 87, 88]. Previous studies have even revealed that more than 60% of developers search for source code every day [30, 80]. With the existence of super-repositories such as GitHub hosting millions of open source projects, there are opportunities to satisfy the search need of developers for resolving a large variety of programming issues.

Oftentimes, developers are looking for code fragments that offer similar functionality than some other code fragments. For example, a developer may need to find Java implementations of all sorting algorithms that could be more efficient than the one she/he has. We refer to such code fragments which have similar functional behavior even if their code is dissimilar as *semantic clones*. The literature also refers to them as *Type-4* clones for consistency with the taxonomy of code clones [13, 73]. Besides the potential of helping developers collect relevant examples to improve their code, finding similar code fragments is an important endeavor, at they can provide essential ingredients for addressing challenges in various software engineering techniques. Among such challenges, we can enumerate automated software transplantation [9], software diversification [11], and even software repair [41].

Finding semantically similar code fragments is, however, challenging to perform statically, an essential trait to ensure scalability. A few studies [21, 35] have investigated program inputs and outputs to find equivalent code fragments. More recently, Su et al. [81] have proposed an approach to find code relatives relying on instruction-level execution traces (i.e., code with similar execution behaviors). Unfortunately, all such dynamic approaches cannot scale to large repositories because of their requirement of runtime information.

Our key insight to statically find code fragments which are semantically similar is first to undertake a description of the functionality implemented by any code fragment. Then, such descriptions can be used to match other code fragments that could be described similarly. This insight is closely related to the work by Marcus and Maletic on *high-level concept clones* [62] whose detection is based on source code text (comments and identifiers) providing an abstract view of code. Unfortunately, their approach can only help to identify high-level concepts (e.g., abstract data types), but is not targeted at describing functionalities per se.

Because of the vocabulary mismatch problem [24, 29, 89, 90] between code terms and human description words, it is challenging to identify the most accurate terms to summarize, in natural language, the functionality implemented by a code fragment.

To work around the issue of translating a code fragment into natural language description terms, one can look[2] up to a developer community. Actually, developers often resort to web-based resources such as blogs, tutorial pages, and Q&A sites. StackOverflow is one of such leading discussion platforms, which has gained popularity among software developers. In StackOverflow, 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. StackOverflow provides social mechanisms to assess and improve the quality of posts that leads implicitly to high-quality source code snippets on the one hand as well as concise and comprehensive questions on the other hand. Our intuition is that information in Q&A sites can be leveraged as a collective knowledge to build an

*Corresponding authors.

[2] Because software projects can be more or less poorly documented, we do not consider source code comments as a reliable and consistent database for extracting descriptions. Instead, we rely on developer community Q&A sites to collect descriptions.

ICSE '18, May 27-June 3, 2018, Gothenburg, Sweden

ACM ISBN 978-1-4503-5638-1/18/05…$15.00
https://doi.org/10.1145/3180155.3180187

intermediate translation step before the exploration of large code bases.

This paper. We propose FaCoY (Find a Code other than Yours) as a novel, static, scalable and effective code-to-code search engine for finding semantically similar code fragments in large code bases.

Overall, we make the following contributions:

- **The FaCoY approach for code-to-code search**: We propose a solution to discover code fragments implementing similar functionalities. Our approach radically shifts from mere syntactic patterns detection. It is further fully static (i.e., relies solely on source code) with no constraint of having runtime information. FaCoY is based on *query alternation*: after extracting structural code elements from a code fragment to build a query, we build alternate queries using code fragments that present similar descriptions to the initial code fragment. We instantiate the FaCoY approach based on indices on Java files collected from GitHub and Q&A posts from `StackOverflow` to find the best descriptions of functionalities implemented in a large and diversified set of code snippets.
- **A comprehensive empirical evaluation**. We present evaluation results demonstrating that FaCoY can accurately help search for (syntactically and semantically) similar code fragments, outperforming popular online code-to-code search engines. We further show, with the BigCloneBench benchmark [83], that we perform better than the state-of-the-art on static code clone detectors identifying semantic clones; our approach identifies over 635,000 semantic clones, while others detect few to no semantic clones. We also break down the performance of FaCoY to highlight the added-value of our *query alternation* scheme. Using the DyCLINK dynamic tool [81], we validate that, in 68% of the cases, our approach indeed finds code fragments that exhibit similar runtime behavior. Finally, we investigate the capability of FaCoY to be leveraged for repair patch recommendation.

2 MOTIVATION AND INSIGHT

Finding similar code fragments beyond syntactic similarities has several uses in the field of software engineering. For example, developers can leverage a code-to-code search tool to find alternative implementations of some functionalities. Recent automated software engineering research directions for software transplantation or repair constitute further illustrations of how a code-to-code search engine can be leveraged.

Despite years of active research in the field of code search and code clones detection, few techniques have explicitly targeted semantically similar code fragments. Instead, most approaches focus on textually, structurally or syntactically code fragments. The state-of-the-art techniques on static detection of code clones leverage various intermediate representations to compute code similarity. Token-based [8, 39, 56] representations are used in approaches that target syntactic similarity. AST-based [12, 34] representations are employed in approaches that detect similar but potentially structurally different code fragments. Finally, (program dependency) graph-based [49, 57] representations are used in detecting clones where statements are not ordered or parts of the clone code are intertwined with each other. Although similar code fragments identified by all these approaches usually have similar behavior, the contemporary static approaches still miss finding such fragments which have similar behavior even if their code is dissimilar [36].

To find similarly behaving code fragments, researchers have relied upon dynamic code similarity detection which consists in identifying programs that yield similar outputs for the same inputs. State-of-the-art dynamic approaches generate random inputs [35], rely on symbolic [55] or concolic execution [50] and check abstract memory states [45] to compute function similarity based on execution outputs. The most recent state-of-the-art on dynamic clone detection focuses on the computations performed by the different programs and compares instruction-level execution traces to identify equivalent behavior [81]. Although these approaches can be very effective in finding semantic code clones, dynamic execution of code is not scalable and implies several limitations for practical usage (e.g., the need of exhaustive test cases to ensure confidence in behavioral equivalence).

To search for relevant code fragments, users turn to online code-to-code search engines, such as Krugle [1], which statically scan open source projects. Unfortunately, such Internet-scale search engines still perform syntactic matching, leading to low-quality output in terms of semantic clones.

On the key idea Consider the code fragments shown in Figure 1. They constitute variant implementations for computing the hash of a string. These examples are reported in BigCloneBench [83] as type-4 clones (i.e., semantic clones). Indeed, their syntactic similarity is limited to a few library function calls. Textually, only about half of the terms are similar in both code fragments. Structurally, the first implementation presents only one execution path while the second includes two paths with the try/catch mechanism.

```
public String getHash(final String password)
        throws NoSuchAlgorithmException, UnsupportedEncodingException {
    final MessageDigest digest = MessageDigest.getInstance("MD5");
    byte[] md5hash;
    digest.update(password.getBytes("utf-8"), 0, password.length());
    md5hash = digest.digest();
    return convertToHex(md5hash);
}
```

(a) Excerpt from `MD5HashHelperImpl.java` in the `yes-cart` project.

```
public static String encrypt(String message) {
    try {
        MessageDigest digest = MessageDigest.getInstance("MD5");
        digest.update(message.getBytes());
        BASE64Encoder encoder = new BASE64Encoder();
        return encoder.encode(digest.digest());
    } catch (NoSuchAlgorithmException ex) {
    ...
```

(b) Excerpt from `Crypt.java` in the `BettaServer` project.

Figure 1: Implementation variants for hashing.

To statically identify the code fragments above as semantically similar, a code-to-code search engine would require extra hint that (i) *getHash* and *encrypt* deal with related concepts and that (ii) *BASE64Encoder* API is offering similar functionality as *convertToHex*. Such hints can be derived automatically if one can build a comprehensive collection of code fragments with associated descriptions allowing for high-level groupings of fragments (based on natural language descriptions) to infer relationships among code tokens. The inference of such relationships will then enable the generation of **alternate queries** displaying related, but potentially syntactically different, tokens borrowed from other code fragments having similar descriptions than the input fragment. Thus, given the code example of Figure 1(a), the system will detect similar code fragments by matching not only tokens that are syntactically[3] similar to the ones in this code fragment (i.e., `gethash`, `messagedigest`, and `converttohex`), but also others similar to known related tokens (e.g., `base64encoder`, `encrypt`, etc.). Such a system would then be able to identify the code fragment of Figure 1(b) as a semantically similar code fragment (i.e., a semantic clone).

We postulate that Q&A posts and their associated answers constitute a comprehensive dataset with a wealth of information on how different code tokens (found in code snippets displayed as example answers) can be matched together based on natural language descriptions (found in questions). Figure 2 illustrates the steps that could be unfolded for exploiting Q&A data to find semantic clones in software repositories such as Github.

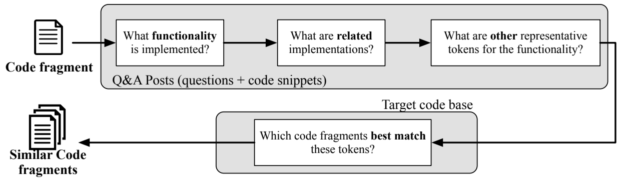

Code fragment
What **functionality** is implemented?
What are **related** implementations?
What are **other** representative tokens for the functionality?
Q&A Posts (questions + code snippets)
Target code base
Which code fragments **best match** these tokens?
Similar Code fragments

Figure 2: Conceptual steps for our search engine.

Given a code fragment, the first step would consist to infer natural language terms that best describe the functionality implemented. To that end, we must match the code fragment with the closest code example provided in a Q&A post. Then, we search in the Q&A dataset all posts with similar descriptions to collect their associated code examples. By mixing all such code examples, we can build a more diverse set of code tokens that could be involved in the implementation of the relevant functionality. Using this set of tokens, we can then search real-world projects for fragments which may be syntactically dissimilar while implementing similar functionality.

Basic definitions

We use the following definitions for our approach in Section 3.
• **Code Fragment**: A contiguous set of code lines that is fed as input to the search engine. The output of the engine is also a list of code fragments. We formalize it as a finite sequence of tokens representing (full or partial) program source code at different granularities: e.g., a function, or an arbitrary sequence of statements.
• **Code Snippet**: A code fragment found in Q&A sites. We propose this terminology to differentiate code fragments that are leveraged during the search process from those that are used as input or that are finally yielded by our approach.
• **Q&A Post**: A pair $p = (q, A)$ also noted p_A^q, where q is a question and A is a list of answers. For instance, for a given post p_A^q, question q is a document describing what the questioner is trying to ask about and $a \in A$ is a document that answers the question in q. Each answer a can include one or several code snippets: $S = snippets(a)$, where S is a set of code snippets.

We also recall for the reader the following well-accepted definitions of clone types [13, 73, 81]:
•**Type-1**: Identical code fragments, except for differences in white-space, layout, and comments.
•**Type-2**: Identical code fragments, except for differences in identifier names and literal values, in addition to Type-1 clone differences.
•**Type-3**: Syntactically similar code fragments that differ at the statement level. The fragments have statements added, modified and/or removed with respect to each other, in addition to Type-1 and Type-2 clone differences.
•**Type-4**: Syntactically dissimilar code fragments that implement the same functionality. They are also known as *semantic clones*.
Disclaimer. In this work, we refer to a pair of code fragments which are semantically similar as *semantic clones*, although they might have been implemented independently (i.e., no cloning, e.g., copy/-paste, was involved). Such pairs are primary targets of FaCoY.

3 APPROACH

FaCoY takes a code fragment from a user and searches in a software projects' code base to identify code fragments that are similar to the user's input. Although the design of FaCoY is targeted at taking into account functionally similar code with syntactically different implementations, the search often returns fragments that are also syntactically similar to the input query.

Figure 3 illustrates the steps that are unfolded in the working process of the search:

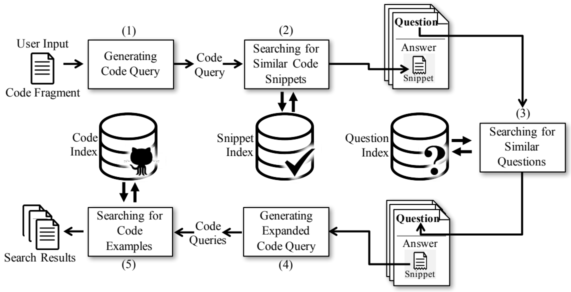


Figure 3: Overview of FaCoY.

(1) When FaCoY receives a code fragment, it generates a structured query called *Code Query* based on the code elements present in the fragment (Section 3.2.1).
(2) Given a code query, FaCoY searches for Q&A posts that include the most syntactically similar code snippets. To that end, the query is matched against the *Snippet Index* of Q&A posts (Section 3.2.2).
(3) Once the relevant posts are identified, FaCoY collects natural language descriptive terms from the associated questions and matches them with the *Question index* of Q&A posts to find other relevant posts. The objective is to find additional code snippets that could implement similar functionalities with a diversified set of syntactic tokens (Section 3.2.3).
(4) Using code snippets in answers of Q&A posts collected by previous steps, FaCoY generates code queries that each constitutes an alternative to the first code query obtained from user input in Step (1) (Section 3.2.4).
(5) As the final step, FaCoY searches for similar code fragments by matching the code queries yielded in Step (4) against the *Code Index* built from the source code of software projects (Section 3.2.5).

[3] Naive syntactic matching may lead to false positives. Instead, there is a need to maintain qualification information about the tokens (e.g., the token represents a method call, a literal, etc.) cf. Section 3.

3.1 Indexing

Our approach constructs three indices, *snippet, question, and code*, in advance to facilitate the search process as well as ensuring a reasonable search speed [61]. To create these indices, we use Apache Lucene, one of the most popular information retrieval libraries [64]. Lucene indices map tokens into instances which in our cases can be natural language text, code fragments or source code files.

3.1.1 Snippet Index.

The Snippet Index in FaCoY maintains the mapping information between answer document IDs of Q&A posts and their code snippets. It is defined as a function: $Inx_S : S \rightarrow 2^P$, where S is a set of code snippets, and P is a set of Q&A posts. 2^P denotes the power set of P (i.e., the set of all possible subsets of P). This index function maps a code snippet into a subset of P, in which the answer in a post has a similar snippet to the input. Our approach leverages the Snippet Index to retrieve the Q&A post answers that include the most similar code snippets to a given query.

To create this index, we must first collect code examples provided as part of a Q&A post answer. Since code snippets are mixed in the middle of answer documents, it is necessary to identify such regions containing the code snippets. Fortunately, most Q&A sites, including StackOverflow, make posts available in a structured document (e.g., HTML or XML) and explicitly indicate source code elements with ad-hoc tags such as <code> · · · <code> allowing FaCoY to readily parse answer documents and locate code snippets.

After collecting a code snippet from an answer, FaCoY creates its corresponding index information as a list of index terms. An index term is a pair of the form '**token_type**:*actual_token*" (e.g., **used_class**:*ActionBar*). Table 1 enumerates examples of token types considered by FaCoY. The complete list is available in [23].

Table 1: Examples of token types for snippet index creation.

Type	Description
typed_method_call	(Partially) qualified name of called method
unresolved_method_call	Non-qualified name of called method
str_literal	String literal used in code

Figure 4 shows an example of code fragment with the corresponding index terms.

```
public class BaseActivity extends AppCompatActivity{
    public static final int IMAGE_PICK_REQUEST_CODE = 5828;
    public static final int MUSIC_PICK_REQUEST_CODE = 58228;
    protected ActionBar actionBar;
    @Override
    protected void onCreate(Bundle savedInstanceState){
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_based);
} }
```

(a) Example code fragment.

```
extends:AppCompatActivity
used_class:BaseActivity
used_class:R
used_class:R.layout
used_class:ActionBar
used_class:Bundle
methods_declaration:onCreate
typed_method_call:AppCompatActivity.onCreate
typed_method_call:AppCompatActivity.setContentView
```

(b) Corresponding index terms.

Figure 4: Extraction of index terms from a code fragment.

To generate index terms, FaCoY must produce the abstract syntax tree (AST) of a code snippet. Each AST node that corresponds to a token type in Table 1[4] is then retained to form an index term. Finally, FaCoY preserves, for each entry in the index, the answer document identifier to enable reverse lookup.

The main challenge in this step is due to the difficulty of parsing *incomplete code*, a common trait of code snippets in Q&A posts [79]. Indeed, it is common for such code snippets to include partial statements or excerpts of a program, with the purpose to give only some key ideas in a concise manner. Often, snippets include ellipses (i.e., "...") before and after the main code blocks. To allow parsing by standard Java parsers, FaCoY resolves the problem by removing the ellipses and wrapping code snippets with a custom dummy class and method templates.

Besides incompleteness, code snippets present another limitation due to *unqualified names*. Indeed, in code snippets, enclosing class names of method calls are often ambiguous [20]. A recent study [82] even reported that unqualified method names are pervasive in code snippets. Recovering unqualified names is, however, necessary for ensuring accuracy when building the Snippet Index. To that end, FaCoY transforms unqualified names to (partially) qualified names by using structural information collected during the AST traversal of a given code snippet. This process converts variable names on which methods are invoked through their corresponding classes. Figure 5 showcases the recovering of name qualification information. Although this process cannot recover all qualified names, it does improve the value of the Snippet Index.

```
SAXParserFactory ft;
InputSource is ;
URL ul = new URL(feed)
ft = SAXParserFactory.newInstance();
SAXParser pr = factory.newSAXParser();
XMLReader rd = pr.getXMLReader();
RssHandler hd = new RssHandler();
rd.setContentHandler(hd);
is = new InputSource(url.openStream());
xmlreader.parse(is);
return hd.getFeed();
```

(a) Fragment before recovering name qualification.

```
SAXParserFactory ft;
InputSource is;
URL ul = new URL(feed)
ft = SAXParserFactory.newInstance();
SAXParser pr = _SAXParserFactory_.newSA...
XMLReader rd = _SAXParser_.getXMLReader();
RssHandler hd = new RssHandler();
_XMLReader_.setContentHandler(hd);
is = new InputSource(_URL_.openStream());
_XMLReader_.parse(is);
return _RssHandler_.getFeed();
```

(b) Fragment after recovering name qualification.

Figure 5: Recovery of qualification information [79]. Recovered name qualifications are highlighted by red color.

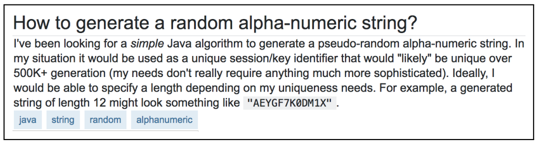
Disclaimer. FaCoY is compliant with the Creative Commons Attribute-ShareAlike license [4, 19] of StackOverflow: we do not redistribute any code from Q&A posts. We only mine developer code examples in StackOverflow to learn relationships among tokens.

3.1.2 Question Index.

The Question Index maps a set of word tokens into Q&A posts. The mapping information serves to identify Q&A posts where the questions are similar to the questions retrieved in Step (2) (cf. Figure 3) whose answers contain code snippets that are similar to the user input. FaCoY leverages this index to construct alternate queries to the one derived from user input. These alternate queries are necessary to increase the opportunities for finding semantically similar code fragments rather than only syntactically similar fragments. The following equation defines the Question Index function: $Inx_Q : Q \rightarrow 2^P$, where Q is a set of questions, and P is a set of Q&A posts. This function maps a set of words from the input into a subset of posts ($P_E \in P$) that are similar to the input.

[4] Our approach focuses on the types in the table since they represent most of the characteristics in a code snippet.

To build the Question Index, FaCoY takes the question part (q) of each Q&A post and generates index terms. To that end a pre-processing of the question text is necessary. This pre-processing includes tokenization (e.g., splitting camel case), stop word removal[5] [61], and stemming. From the pre-processing output, FaCoY builds index terms in the form of "**term**:*token*". Each index term is further mapped with the question where its token is originated from, to keep an inverse link. Figure 6 illustrates how, given a question text, index terms are generated.



How to generate a random alpha-numeric string?

I've been looking for a *simple* Java algorithm to generate a pseudo-random alpha-numeric string. In my situation it would be used as a unique session/key identifier that would "likely" be unique over 500K+ generation (my needs don't really require anything much more sophisticated). Ideally, I would be able to specify a length depending on my uniqueness needs. For example, a generated string of length 12 might look something like `"AEYGF7K0DM1X"`.

(a) Description text in a question.

term:simpl, term:java, term:algorithm, term:generat, term:pseudo, term:random, term:alpha, term:numer, term:string, term:situat, term:uniq, term:session, term:key, term:identifi, term:uniq, term:500k, term:requir, term:sophist, term:ideal, term:length, term:depend, term:string, term:length, term:12, term:aeygf7k0dm1x

(b) Resulting index terms generated from (a).

Figure 6: Example of question index creation.

3.1.3 Code Index.

The *Code Index* maintains mapping information between tokens and source code files. FaCoY leverages this index to search for code examples corresponding to a code query yielded at the end of Step (4) (cf. Figure 3). This index actually defines the search space of our approach (e.g., F-droid repository of Android apps, or Java projects in Github, or a subset of Mozilla projects). The Code Index function is defined as: $Inx_C : S \to 2^F$, where S is a set of code snippets and F is a set of code fragments. F actually defines the space of FaCoY.

The creation process of the Code Index is similar to the process for building the Snippet Index that is described in Section 3.1.1. FaCoY first scans all available source code files in the considered code base. Then, each file is parsed[6] to generate an AST from which FaCoY collects the set of AST nodes corresponding to the token types listed in Table 1. The AST nodes and their actual tokens are used for creating index terms in the same format as in the case of the Snippet Index. Finally, each index term is mapped to the source code file where the token of the term has been retrieved.

3.2 Search

Once the search indices are built, FaCoY is ready to take a user query and search for relevant code fragments. Algorithm 1 formulates the search process for a given user query. Its input also considers the three indices described in Section 3.1 and stretch parameters used in the algorithm. The following sections detail the whole process.

3.2.1 Generating a Code Query from a User Input.

As the first step of code search, FaCoY takes a user input and generates a code query from the input to search the snippet index (Line 2 in Algorithm 1). The code query is in the same form of index terms illustrated in Figure 4 so that it can be readily used to match the index terms in the Snippet Index.

Algorithm 1: Similar code search in FaCoY.

```
Input : $c$: code fragment (i.e., user query).
Input : $Inx_S(q_s)$: a function of a code snippet, $s$, to a sorted list of posts, $P_s$.
Input : $Inx_Q(q_q)$: a function of a question, $q$, to a sorted lis of questions, $Q_q$.
Input : $Inx_C(q_s)$: a function of a code snippet, $s$, to a sorted list of code
        fragments, $F_s$.
Input : $n_s, n_q, n_c$: stretch parameters for snippet, question, and code search,
        respectively (e.g., consider Top $n$ similar snippets).
Output: $F$: a set of code fragments that are similar to $c$.
1  Function SearchSimilarCodeExamples($c, Inx_S, Inx_Q, Inx_C, n_s, n_q, n_c$)
2      $q_{in} \Leftarrow$ genCodeQuery($c$);
3      $P_s \Leftarrow Inx_S(q_{in})$.top($n_s$);
4      $Q_s \Leftarrow P_s$.foreach($p_i$ => $takeQuestion(p_i)$);
5      $P_e \Leftarrow Q_s$.foreach($q_i$ => $Inx_Q(q_i)$.top($n_q$));
6      let $F \Leftarrow \emptyset$;
7      foreach $p_i \in P_e$ do
8          $s \Leftarrow takeSnippet(getAnswer(p_i))$;
9          $q_{ex} \Leftarrow$ genCodeQuery($s$);
10         $F \Leftarrow F \cup Inx_C(q_{ex})$.top($n_c$));
11     end
12     return $F$;
13 end
```

To generate a code query, our approach follows the process described in Section 3.1.1 for generating the index terms of any given code snippet. If the user input is also an incomplete code fragment (i.e., impossible to parse), FaCoY seamlessly wraps the fragment using a dummy class and some method templates after removing ellipses. It then parses the code fragment to obtain an AST and collect the necessary AST nodes to generate index terms in the form of **token_type**:*actual_token*.

3.2.2 Searching for Similar Code Snippets.

After generating a code query from a user input, our approach tries to search for similar snippets in answers of Q&A posts (Line 3 in Algorithm 1). Since the code query and index terms in the snippet index are in the same format, our approach uses full-text search (i.e., examining all index terms for a code snippet to compare with those in a code query). The full-text function implemented by Lucene is utilized.

Our approach computes rankings of the search results based on a scoring function that measures the similarity between the code query and matched code snippets. FaCoY integrates two scoring functions, Boolean Model (BM) [51] and Vector Space Model (VSM) [76], which are already implemented in Lucene. First, BM reduces the amount of code snippets to be ranked. Our approach transforms the code query of the previous step, q_in, into a normal form and matches code snippets indexed in the snippet index. We adopt best match retrieval to find as many similar snippets as possible. Then, for the retained snippets, VSM computes similarity scores. After computing TF-IDF (Term Frequency - Inverse Document Frequency) [75] of terms in each snippet as a weighting scheme, it calculates Cosine similarity values between the code query and indexed snippets.

From the sorted list of similar snippets, FaCoY takes top n_s snippets (i.e., those that will allow to consider only most relevant natural language descriptions to associate with the user input). By default, in all our experiments in this paper, unless otherwise indicated, we set the value of n_s (stretch parameter) to 3.

3.2.3 Searching for Similar Questions.

In this step (Line 4 in Algorithm 1), our approach searches for questions similar to

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

[6] This step also recovers qualified names by applying, whenever necessary, the same procedure described in Section 3.1.1.

the questions of Q&A posts retrieved in the previous step (cf. Section 3.2.2). The result of this step is an additional set of Q&A posts containing questions that are similar to the given questions identified as describing best the functionality implemented in the user input. Thanks to this **search space enrichment** approach, FaCoY can include more diverse code snippets for enumerating more code tokens which are semantically relevant.

To search for similar questions, we use the Question Index described in Section 3.1.2. Since all questions are indexed beforehand, the approach simply computes similarity values between questions as the previous step does (cf. Section 3.2.2), i.e., filtering questions based on BM and calculating cosine similarity based on VSM.

Once similarity scores are computed, we select the top n_q posts based on the scores of their questions, as the goal is to recommend most relevant questions rather than listing up all similar questions. Since it takes n_s posts for each of n_q questions retrieved in Line 3 of Algorithm 1, the result of this step consists of $n_s \times n_q$ posts when using the same stretch parameter for both steps. FaCoY can be tuned to consider different stretch values for each step.

3.2.4 **Generating Alternate Code Queries.** This step (Line 5 in Algorithm 1) generates code queries from code snippets present in newly retrieved Q&A posts at the end of the previous step (cf. Section 3.2.3). Our approach in this step first takes Q&A posts identified in Line 4 and extracts code snippets from their answer parts. It then follows the same process described in Section 3.2.1 to generate code queries. Since the result of the previous step (Line 4) is $n_s \times n_q$ posts (when using the same value for stretch parameters), this step generates at most $n_s \times n_q$ code queries, referred to as *alternate queries*.

3.2.5 **Searching for Similar Code fragments.** As the last step, FaCoY searches the Code Index for similar code fragments to output (Lines 6–12 in Algorithm 1). Based on the alternate code queries generated in the previous step (cf. Section 3.2.4), and since code queries and index terms are represented in the same format, FaCoY can leverage the same process of Step (2) illustrated in Section 3.2.2 to match code fragments. While the step described in Section 3.2.2 returns answers containing code snippets similar to a user query, the result of this step is a set of source code files containing code fragments corresponding to the alternate code query from the previous step. Note that FaCoY provides at most $n_s \times n_q \times n_c$ code fragments as Line 10 in Algorithm 1 uses n_c to take top results.

Delimitating code fragments: Since displaying the entire content of a source code file will be ineffective for users to readily locate the identified similar code fragment, FaCoY specifies a code range after summarizing the content [61]. To summarize search results into a specific range, FaCoY uses a query-dependent approach that displays segments of code based on the query terms occurring in the source file. Concretely, the code fragment starts from k lines preceding the first matched token and spreads until k lines following the last matched token.

4 EVALUATION

In this section, we describe the design of different assessment scenarios for FaCoY and report on the evaluation results. Specifically, our experiments aim to address the following research questions:

- **RQ1**: How relevant are code examples found by FaCoY compared to other code-to-code search engines?
- **RQ2**: What is the effectiveness of FaCoY in finding semantic clones based on a code clone benchmark?
- **RQ3**: Do the semantically similar code fragments yielded by FaCoY exhibit similar runtime behavior?
- **RQ4**: Could FaCoY recommend correct code as alternative of buggy code?

To answer these research questions, we build a prototype version of FaCoY where search indices are interchangeable to serve the purpose of each assessment scenario. We provide in Section 4.1 some details on the implementation before describing the design and results for the different evaluations.

4.1 Implementation details

Accuracy and speed performance of a search engine are generally impacted by the quantity of data and the quality of the indices [70]. We collect a comprehensive dataset from `GitHub`, a popular and very large open source project repository, as well from `StackOverflow`, a popular Q&A site with a large community to curate and propose accurate information on code examples. We further leverage the Apache Lucene library, whose algorithms have been tested and validated by researchers and practitioners alike for indexing and searching tasks.

For building the Code Index representing the search space of the code base where to code fragments, we consider the `GitHub` repository. We focus on Java projects since Java remains popular in the development community and is associated with a large number of projects in `GitHub` [14]. Overall, we have enumerated 2,993,513 projects where Java is set as the main language. Since there are many toy projects on `GitHub` [38], we focused on projects that have been forked at least once by other developers and dropped out projects where the source code include non-ascii characters. Table 2 summarizes the collected data.

Table 2: Statistics on the collected GitHub data.

Feature	Value	Feature	Value
Number of projects	10,449	Number of duplicate files	382,512
Number of files	2,163,944	LOCs	>384M

For building the Snippet and Question indices, we downloaded a dump file from the `StackOverflow` website containing all posts between July 2008 and September 2016 in XML format. In total, we have collected and indexed 1,856,592 posts tagged as about Java or Android coding. We have used a standard XML parser to extract natural language elements (tagged with `<p>. . . </p>` markers) and code snippets (tagged with `<code>. . . </code>`). It should be noted that we first filter in code snippets from answers that have been accepted by the questioner. Then we only retained those accepted answers that have been up-voted at least once. These precautions aim at ensuring that we leverage code snippets that are of high quality and are really matching the questions. As a result, we eventually used 268,264 Q&A posts to build the snippet and question indices. By default, we set all three stretch parameters to $n_s = n_q = n_c = 3$. The stretch for delimitating output code fragments is also set to $k = 3$.

4.2 RQ1: Comparison with code search engines

Design: In this experiment, we compare the search results of FaCoY with those from online code search engines. We focus on Krugle [1] and searchcode [2] since these engines support code-to-code search. As input code fragments, we consider code examples implementing popular functionalities that developers ask about. To that end, we select snippets from posts in StackOverflow. The snippets are selected following two requirements: (1) the associated post is related to "Java" and (2) the answer include code snippets. We select code snippets in the top 10 posts with the highest view counts (for their questions). Table 3 lists the titles of StackOverflow posts whose code snippets are used in our experiment. Note that, for a fair comparison and to avoid any bias towards FaCoY, the actual posts (including the code snippets in their answers) shown in the table are removed from the snippet and question indices; this prevents our tool from leveraging answer data in advance, which would be unfair.

Table 3: Top 10 StackOverflow Java posts with code snippets.

Query #	Question title
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?

Figure 7 shows an example of input code fragments collected from StackOverflow that is used in our evaluation. 10 code snippets[7] are then used to query FaCoY, Krugle, and searchcode.

```
import java.security.SecureRandom;
import java.math.BigInteger;
public final class SessionIdentifierGenerator {
  private SecureRandom random = new SecureRandom();
  public String nextSessionId() {
    return new BigInteger(130, random).toString(32);
  }
}
```

Figure 7: Code snippet associated to Q2 in Table 3.

On each search engine, we consider at most the top 20 search results for each query and manually classify them into one of the four clone types defined in Section 2.

Table 4: Statistics based on manual checks of search results.

	FaCoY				Searchcode			Krugle	
Query	**# outputs**	**Type-2**	**Type-3**	**Type-4**	**# outputs**	**Type-1**	**Type-3**	**# outputs**	**Type-1**
Q1	18		5(27.7%)	4(22.2%)	0			0	
Q2	21		6(28.5%)	2(9.5%)	0			0	
Q3	18		9(50%)		0			0	
Q4	0				20	20(100%)		1	1(100%)
Q5	19	1(5.2%)	2(10.5%)	6(31.5%)	3	2(66.6%)	1(33.3%)	0	
Q6	9	1(11.1%)	3(30%)	1(11.1%)	20	20(100%)		0	
Q7	17				0			0	
Q8	17		7(41.1%)	7(41.1%)	0			0	
Q9	0				2		2(100%)	0	
Q10	9		1(11.1%)	7(77.7%)	0			0	

Result: Table 4 details statistics on the search results for the different search engines. FaCoY, Krugle, and searchcode produce search results for eight, four and one queries respectively. Search results can also be false positives. We evaluate the efficiency of FaCoY using the *Precision@k* metric defined as follows:

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

where $relevant_{i,k}$ represents the relevant search results for query i in the top k returned results, and Q is a set of queries.

FaCoY achieves 57.69% and 48.82% scores for *Precision*@10 and *Precision*@20 respectively.

We further categorize the true positive code fragments based on the clone type. Krugle appears to be able to identify only Type-1 clones. searchcode on the other hand also yields some Type-3 code clones for 2 queries. Finally, FaCoY mostly successfully retrieves Type-3 and Type-4 clones.

> Unlike online code-to-code search engines, FaCoY can identify (1) similar code fragments for a more diverse set of functionality implementations. Those code fragments can be syntactically dissimilar to the query while implementing similar functionalities.

4.3 RQ2: Finding similar code in IJaDataset

Design: This experiment aims at evaluating FaCoY against an existing benchmark. Since our code-to-code search engine is similar to a code clone detector in many respects, we focus on assessing which clones FaCoY can effectively identify in a code clone benchmark. A clone benchmark contains pairs of code fragments which are similar to each other.

We leverage BigCloneBench [83], one of the biggest (8 million clone pairs) code clone benchmarks publicly available. This benchmark is built by labeling of pairs of code fragments from the IJaDataset-2.0 [3]. IJaDataset includes approximately 25,000 open-source Java projects consisting of 3 million source code files and 250 millions of lines of code (MLOC). BigCloneBench maintainers have mined this dataset focusing on a specific set of functionalities. They then record metadata information about the identified code clone pairs for the different functionalities. In this paper, we use a recent snapshot of BigCloneBench including clone pairs clustered in 43 functionality groups made available for the evaluation of SourcererCC [74].

We consider 8,345,104 clone pairs in BigCloneBench based on the same criteria used in [74]: both code fragments in a clone pair have at least 6 lines and 50 tokens in length, a standard minimum clone size for benchmarking [13, 84].

Clone pairs are further assigned a type based on the criteria in [74]: Type-1 and Type-2 clone pairs are classified according to the classical definitions recalled in Section 2. Type-3 and Type-4 clones are further divided into four sub-categories based on their syntactical similarity: Very Strongly Type 3 (VST3), Strongly Type 3 (ST3), Moderately Type 3 (MT3), and Weakly Type 3/Type 4 (WT3/4). Each clone pair (unless it is Type 1 or 2) is identified as one of four if its similarity score falls into a specific range; VST3: [90%, 100%), ST3: [70%, 90%), MT3: [50%, 70%), and WT3/4: [0%, 50%).

For this experiment, we adapt the implementation described in Section 4.1. Since the experiment conducted in [74] detected clones only from IJaDataset, the GitHub-based code index in our tool is replaced by a custom index generated from IJaDataset for a fair comparison. This makes FaCoY search only code fragments in IJaDataset. In addition, the stretch parameters (see Algorithm 1)

[7] Code snippets available on project web page [23].

are set to $n_s = n_q = 3, n_c = 100$, making FaCoY yield as many snippets, posts and fragments as possible in each step.

We feed FaCoY with each code fragment referenced in the benchmark in order to search for their clones in the IJaDataset. We compare each pair, formed by an input fragment and a search result, against the clone pairs of BigCloneBench. We then compute the recall of FaCoY following the definition proposed in the benchmark [83]:

$$Recall = \frac{D \cap B_{tc}}{B_{tc}} \tag{2}$$

where B_{tc} is the set of all true clone pairs in BigCloneBench, and D is the set of clone pairs found by FaCoY.

To quantify the improvement brought by the two main strategies proposed in this work, namely *query alternation* and *query structuring*, we define four different search engine configurations:

- Baseline SE: The baseline search engine does not implement any query structuring or query alternation. Input code query, as well as the search corpus, are treated as natural language text documents. Search is then directly performed by matching tokens with no type information.
- FaCoY$_{noQA}$: In this version, only query structuring is applied. No query alternation is performed, and thus only input code query is used to match the search corpus.
- FaCoY$_{noUQ}$: In this version, query alternation is performed along with query structuring, but initial input query is left out.
- FaCoY: This version represents the full-feature version of the code-to-code search engine: queries are alternated and structured, and initial input code query also contributes in the matching process.

Result: Table 5 details the recall[8] scores for the baseline SE, FaCOY$_{noQA}$, FaCOY$_{noUQ}$ and FaCoY. Recall scores are summarized per clone type with the categories introduced above. Since we are reproducing for FaCoY the experiments performed in [74], we directly report in this table all results that the authors have obtained on the benchmark for state-of-the-art Nicad [18], iClones [28], SourcererCC [74], CCFinderX [39], Deckard [34] clone detectors.

Table 5: Recall scores on BigCloneBench **[83].**

	Clone Types					
	T1	T2	VST3	ST3	MT3	WT3/T4
(# of Clone Pairs)	(39,304)	(4,829)	(6,171)	(18,582)	(90,591)	(6,045,600)
FaCoY	65	90	67	69	**41**	**10** (635,844)★
FaCoY $_{noUQ}$	35	74	45	55	37	10
FaCoY $_{noQA}$	66	26	56	54	20	2
Baseline SE	66	26	54	50	20	2
SourcererCC	**100**	98	93	61	5	0
CCFinderX†	**100**	93	62	15	1	0
Deckard†	60	58	62	31	12	1
iClones†	**100**	82	82	24	0	0
NiCad†	**100**	**100**	**100**	**95**	1	0

★ Cumulative number of WT3/4 clones that FaCoY found.
† The tools could not scale to the entire files of IJaDataset [3].

We recall that FaCoY is a code-to-code search engine and thus the objective is to find semantic clones (i.e., towards Type-4 clones). Nevertheless, for a comprehensive evaluation of the added value of the strategies implemented in the FaCoY approach, we provide comparison results of recall values across all clone types.

Overall, FaCoY produces the highest recall values for moderately Type-3 as well as Weakly Type-3 and Type-4 clones. The recall performance of FaCoY for MT3 clones is an order of magnitude higher than that of 4 out the 5 detection tools. While most tools detect little to no WT3/T4 code clone pairs, FaCoY is able to find over 635,000 clones in the IJADataset. Furthermore, apart from SourcererCC, the other tools could not cover the entire IJaDataset as reported in [74].

Benefit of query structuring. The difference of performance between Baseline SE and FaCOY$_{noQA}$ indicates that query structuring helps to match more code fragments which are not strictly, syntactically, identical to the query (cf. VST3 & ST3).

Benefit of query alternation. The difference of performance between FaCoY and FaCOY$_{noQA}$ definitively confirms that query alternation is the strategy that allows collecting semantic clones: recall for WT3/T4 goes from 2% to 10% and recall for MT3 goes from 20 to 41.

Benefit of keeping input query. The difference of performance between FaCoY and FaCOY$_{noUQ}$ finally indicates that initial input code query is essential for retrieving some code fragments that are more syntactically similar, in addition to semantically similar code fragments matched by alternate queries.

With 10% recall for semantic clones (WT3/T4), FaCoY achieves the best performance score in the literature. Although this score may appear to be small, it should be noted that this corresponds to the identification of 635,844 clones, a larger set than the accumulated set of all clones of other types in the benchmark. Finally, it should also be noted that, while the dataset includes over 7.7 million WT3/T4 code clones, state-of-the-art detection tools can detect only 1% or less of these clones.

We further investigate the recall of FaCoY with regards to the functionalities implemented by clone pairs. In BigCloneBench, every clone pair is classified into one of 43 functionalities, including "Download From Web" and "Decompress zip archive". For each clone type, we count the number of clones that FaCoY can find, per functionality. Functionalities with higher recall tend to have implementations based on APIs and libraries while those with low recall are more computation intensive without APIs. This confirms that FaCoY performs better for programs implemented by descriptive API names since it leverages keywords in snippets and questions. This issue is discussed in Section 5 in detail. Because of space constraints, we refer the reader to the FaCoY project page for more statistics and information details on its performance.

Double-checking FaCoY***'s false positives:*** Although it is one of the largest benchmarks available to the research community, BigCloneBench clone information may not be complete. Indeed, as described in [83], BigCloneBench is built via an incremental additive process (i.e., gradually relaxing search queries) based on keyword and source pattern matching. Thus, it may miss some clones despite the manual verification. In any case, computing precision of a code search engine remains an open problem [74]. Instead, we chose to focus on manually analysing sampled false positives.

We manually verify the clone pairs that are not associated in BigCloneBench, but FaCoY recommended as code clones, i.e., false positives. Our objective is then to verify to what extent they are indeed false positives and not misses by BigCloneBench. We sample 10 false positives per clone type category for a manual check. For 32

[8] It should be noted that Recall is actually Recall@MAX.

out of 60 cases, it turns out that BigCloneBench actually missed to include them. Specifically, it missed 25 Type-4, 2 Type-3, 1 Type-2, and even 4 Type-1 clones. Among the 28 cases, for 26 cases, FaCoY points to the correct file but another location than actual clones. In only two cases FaCoY completely fails. We provide this data in the project web page [23] as a first step towards initiating a curated benchmark of semantic code clones, which can be eventually integrated into BigCloneBench.

FaCoY can find more Type-3, Weakly Type-3 and Type-4 clones than the state-of-the-art, thus fulfilling the objective for which it was designed.

4.4 RQ3: Validating semantic similarity

Design: Since FaCoY focuses on identifying semantically similar code snippets rather than syntactic/structural clones, it is necessary to verify whether the search results of the approach indeed exhibit similar functional behavior (beyond keyword matching with high syntactic differences implied in BigCloneBench). The datasets used in Sections 4.2 and 4.3 are however not appropriate for dynamic analysis: the code must compile as well as execute, and there must be test cases for exercising the programs.

To overcome these challenges, we build on DyCLINK [81], a dynamic approach that computes the similarity of execution traces to detect that two code fragments are relatives (i.e., that they behave (functionally) similarly). The tool has been applied to programs written for *Google Code Jam* [26] to identify code relatives at the granularity of methods. We carefully reproduced their results with the publicly available version of DyCLINK. Among the 642 methods in the code base, DyCLINK matches 411 pairs as code relatives[9]. We consider all methods for which DyCLINK finds a relative and use FaCoY to search for its clones in *codejam*, and we check that the found clones are relatives of the input.

Since FaCoY provides a ranked list of code examples for a given query, we measure the hit[10] ratio of the top N search results. Here, we use the default stretch parameters specified in Section 4.1 and thus $N = 27$. In addition to hit ratio, we compute the Mean Reciprocal Rank (MRR) of the hit cases. To calculate MRR for each clone pair, we use the following formula:

$$MRR = \frac{1}{|Q|} \sum_{i=1}^{|Q|} \frac{1}{rank_i} \qquad (3)$$

where $rank_i$ is the rank position of the corresponding code fragment for the given peer in a clone pair. Q is the number of all queries.

Result: As a result, FaCoY can identify 278 out of 411 code relatives and the hit ratio is 68%. As for efficiency, FaCoY achieves 45% and 88% scores respectively for *Precision*@10 and *Precision*@20, and exhibits an MRR of 0.18, which means FaCoY recommends the code relatives into lower rankings.

On the one hand, since many programs in Google Code Jam often use variables with no meaning (such as `void s(int a){}`), FaCoY cannot find related code in `StackOverflow` and thus cannot build alternate queries, limiting the hit ratio. On the other hand, since DyCLINK also uses a similarity metric to decide on code relativeness, the MRR score of FaCoY could be higher with a more relaxed threshold (currently set at 82%) in DyCLINK.

FaCoY can indeed find alternative fragments that exhibit similar runtime behavior with input code fragment.

4.5 RQ4: Recommending patches with FaCoY

Design: This experiment attempts to use FaCoY to search for correct code that can help fix buggy code. Code search has indeed been proposed recently as a potential step for patch recommendation [25], and even automated repair [40]. Since FaCoY can find code snippets that are semantically similar to a given query code, we conjecture that it can be used for helping find alternative implementations which may turn out to be more correct than the user's code. We assess such a potential application by leveraging the Defects4J benchmark [37].

Defects4J include 395 real bugs: for each bug, the buggy code and the associated fixed version are made available, along with test suites for execution. For each bug, we take buggy code fragment (generally a function) and query FaCoY. By going through the search results from the top, we manually compare each result with the actual paired fixed code. Our objective is to check whether FaCoY's output code fragments can help build a patch that would have been correct w.r.t. to the benchmark fix. We perform the same experiments using `Krugle` [1] and `searchcode` [2].

```
public static boolean equals(CharSequence cs1, CharSequence cs2)
   { return cs1 == null ? cs2 == null : cs1.equals(cs2); }
```

(a) Defects4J buggy code fragment from Commons-LANG†.

```
public static boolean equals(CharSequence a, CharSequence b) {
   if (a == b) return true;
   int length;
   if (a != null && b != null && (length = a.length()) == b.length()) {
      if (a instanceof String && b instanceof String) {
         return a.equals(b);
      }
      for (int i = 0; i < length; i++) {
         if (a.charAt(i) != b.charAt(i)) return false;
      }
      return true;
   }
   return false;
}
```

(b) Code fragment found in GitHub by FaCoY as similar to fragment in (a)*.

```
 public static boolean equals(CharSequence cs1, CharSequence cs2) {
-   return cs1 == null ? cs2 == null : cs1.equals(cs2);
+   if (cs1 == cs2) {
+       return true;
+   }
+   if (cs1 == null || cs2 == null) {
+       return false;
+   }
+   if (cs1 instanceof String && cs2 instanceof String) {
+       return cs1.equals(cs2);
+   }
+   return CharSequenceUtils.regionMatches(...);
 }
```

(c) Actual patch‡ that was proposed to fix the buggy code in (a).

† https://goo.gl/5kn6Zr * https://goo.gl/URdriN ‡ https://goo.gl/PD6KL5

Figure 8: Successful patch recommendation by FaCoY.

For each bug, one of the authors of this paper examined at most top 15 search results from each search engine. When the author marks a result as a good candidate for patch recommendation, two other authors double check, and the final decision is made

[9] We did our best to reproduce the results of DyCLINK. We checked with the authors that the found 411 relatives are consistent with the released tool version.

[10] A "hit" indicates that the corresponding code fragment is in the top N results for a given query.

by majority voting. Note that, since Defects4J projects are also available in GitHub, the fixed code may be in FaCoY corpus. Thus, we have filtered out from the search results any code fragment that is collected from the same project file as the buggy code used as query. Figure 8a shows an example buggy function that we used as query to FaCoY. Fig. 8b shows one of the similar code fragments returned by FaCoY and which we found that it was a good candidate for recommending the patch that was actually applied (cf. Fig. 8c).

Result: Out of 395 bugs in Defects4J, our preliminary results show that FaCoY found similar fixed code examples for 21 bugs. In contrast, `searchcode` located a single code example, while `Krugle` provided no relevant results at all. Specifically, project-specific results are as follows. Lang: 6/65, Mockito: 3/38, Chart: 3/26, Closure: 2/133, Time: 2/27, and Math: 5/106. searchcode was successful only for 1/38 Mokito bug. All details are available in [23].

FaCoY-based search of semantically similar code fragments can support patch/code recommendation, software diversification or transplantation.

5 DISCUSSIONS

Exhaustivity of Q&A data: The main limitation of FaCoY comes from the use of code snippets and natural language descriptions in Q&A posts to enable the generation of alternate queries towards identifying semantically similar code fragments. This data may simply be insufficient with regards to a given user input fragment (e.g., uncommon functionality implementation).

Threats to Validity: As threat to *External validity*, we note that we only used Java subjects for the search. However, the same process can be developed with other programming languages by changing the language parser, the indices for related Q&A posts and project code. Another threat stems from the use of `StackOverflow` and `GitHub` which may be limited. We note however that their data can be substituted or augmented with data from other repositories.

Internal validity: We use subjects from BigCloneBench and Dy-CLINK datasets to validate our work. Those subjects may be biased for clone detection. Nevertheless, these subjects are commonly used and allow for a fair comparison as well as for reproducibility.

6 RELATED WORK

Code search engines. Code search literature is abundant [5, 22, 33, 42, 59, 66, 69, 72, 77, 79]. CodeHow [59] finds code snippets relevant to a user query written in natural language. It explores API documents to identify relationships between query terms and APIs. Sourcerer [5] leverages structural code information from a complete compilation unit to perform fine-grained code search. Portfolio [66] is a code search and visualization approach where a chain of function calls are highlighted as usage scenario. CodeGenie [53, 54] expands queries for interface-driven code search (IDCS). It takes test cases rather than free-form queries as inputs and leverages WordNet and a code-related thesaurus for query expansion. Sirres et al. [79], also use `StackOverflow` data to implement a free-form code search engine.

Clone detection and search. Clone detection has various applications [16, 52, 78] such as plagiarism detection. However, most techniques detect syntactically similar code fragments in source code using tokens [8, 39, 56], AST trees [12, 34], or (program dependency) graphs [49, 57]. Only a few techniques target semantically similar source code clones [35, 45, 47]. Komondoor and Horwitz search for isomorphic sub-graphs of program dependence graphs using program slicing [47]. Jiang and Su compare program execution traces using automated random testing to find functionally equivalent code fragments [35]. MeCC detects semantically-similar C functions based on the similarity of their abstract memory states [45]. White et al. [86] propose to use deep learning to find code clones. Their approach is more effective for Type-1/2/3 clones than Type-4.

Code recommendation systems [31, 32, 65, 71] support developers with reusable code fragments from other programs, or with pointers to blogs and Q&A sites. Strathcona [31] generates queries from user code and matches them against repository examples, Prompter [71] directly matches the current code context with relevant Q&A posts. Although several studies have explored `StackOverflow` posts [10, 25, 60, 68, 79, 85], none, to the best of our knowledge, leveraged `StackOverflow` data to improve clone detection.

Program repair [15, 27, 44] can also benefit from code search. Gao et al. [25] proposed an approach to fix recurring crash bugs by searching for similar code snippets in `StackOverflow`. SearchRepair [40] infers potential patch ingredients by looking up code fragments encoded as SMT constraints. Koyuncu et al. [48] showed that patching tools yield recurrent fix actions that can be explored to fix similar code. Liu et al. [58] explore the potential of fix patterns for similar code fragments that may be buggy w.r.t. FindBugs rules.

API recommendation is a natural application of code search. The Baker approach connects existing source code snippets to API documentation [82]. MUSE [67] builds an index of real source code fragments by using static slicing and code clone detection, and then recommends API usage examples. Keivanloo et al. [43] presented an Internet-scale code search engine that locates working code examples. Buse and Weimer [17] proposed an approach to recommend API usage examples by synthesizing code snippets based on dataflow analysis and pattern abstraction. Bajracharya [7] proposed Structural Semantic Indexing which examines the API calls extracted in source code to determine code similarity.

7 CONCLUSION

We have presented FaCoY, a code-to-code search engine that accepts code fragments from users and recommends semantically similar code fragments found in a target code base. FaCoY is based on **query alternation**: after generating a structured code query summarizing structural code elements in the input fragment, we search in Q&A posts other code snippets having similar descriptions but which may present implementation variabilities. These variant implementations are then used to generate alternate code queries. We have implemented a prototype of FaCoY using `StackOverflow` and `GitHub` data on Java. FaCoY achieves better accuracy than online code-to-code search engines and finds more semantic code clones in BigCloneBench than state-of-the-art clone detectors. Dynamic analysis shows that FaCoY's similar code fragments are indeed related execution-wise. Finally, we have investigated a potential application of FaCoY for code/patch recommendation on buggy code in the Defects4J benchmark.

ACKNOWLEDGEMENTS

We extend our thanks to Seungdeok Han, Minsuk Kim, Jaekwon Lee, and Woosung Jung from Chungbuk National University for their insightful comments on earlier versions of this manuscript. This work was supported by the Fonds National de la Recherche (FNR), Luxembourg, under projects RECOMMEND C15/IS/10449467, FIX-PATTERN C15/IS/9964569, and FNR-AFR PhD/11623818, and by the Japan Society for the Promotion of Science (JSPS) KAKENHI Grant Number JP15H06344. It is also supported by the National Research Foundation of Korea (NRF) grant funded by the Korea government (Ministry of Science, ICT & Future Planning) (No. 2015R1C1A1A01054994).

REFERENCES

[1] 2017. http://krugle.com/. (July. 2017).

[2] 2017. https://searchcode.com/. (July. 2017).

[3] Ambient Software Evoluton Group. 2013. IJaDataset 2.0, http://secold.org/projects/seclone. (Jan. 2013).

[4] Le An, Ons Mlouki, Foutse Khomh, and Giuliano Antoniol. 2017. Stack Overflow: A Code Laundering Platform?. In *Proceedings of the 24th Conference on Software Analysis, Evolution and Reengineering*. IEEE, 283–293.

[5] Sushil Bajracharya, Trung Ngo, Erik Linstead, Yimeng Dou, Paul Rigor, Pierre Baldi, and Cristina Lopes. 2006. Sourcerer: a search engine for open source code supporting structure-based search. In *Companion to the 21st ACM SIGPLAN symposium on Object-oriented programming systems, languages, and applications*. ACM, 681–682.

[6] Sushil Krishna Bajracharya and Cristina Videira Lopes. 2012. Analyzing and mining a code search engine usage log. *Empirical Software Engineering* 17, 4-5 (Aug. 2012), 424–466.

[7] Sushil K. Bajracharya, Joel Ossher, and Cristina V. Lopes. 2010. Leveraging Usage Similarity for Effective Retrieval of Examples in Code Repositories. In *Proceedings of the Eighteenth ACM SIGSOFT International Symposium on Foundations of Software Engineering (FSE '10)*. ACM, New York, NY, USA, 157–166.

[8] B.S. Baker. 1992. A Program for Identifying Duplicated Code. In *Computing Science and Statistics: Proceedings of the 24th Symposium on the Interface*, Vol. 24. 49–57. Issue Mar.

[9] Earl T Barr, Mark Harman, Yue Jia, Alexandru Marginean, and Justyna Petke. 2015. Automated software transplantation. In *Proceedings of the 2015 International Symposium on Software Testing and Analysis*. ACM, 257–269.

[10] Ohad Barzilay, Christoph Treude, and Alexey Zagalsky. 2013. Facilitating crowd sourced software engineering via stack overflow. In *Finding Source Code on the Web for Remix and Reuse*. Springer, 289–308.

[11] Benoit Baudry, Simon Allier, and Martin Monperrus. 2014. Tailored source code transformations to synthesize computationally diverse program variants. In *Proceedings of the 2014 International Symposium on Software Testing and Analysis*. ACM, 149–159.

[12] Ira D Baxter, Andrew Yahin, Leonardo Moura, Marcelo Sant'Anna, and Lorraine Bier. 1998. Clone detection using abstract syntax trees. In *Proceedings of the International Conference onSoftware Maintenance*. IEEE, 368–377.

[13] Stefan Bellon, Rainer Koschke, Giulio Antoniol, Jens Krinke, and Ettore Merlo. 2007. Comparison and evaluation of clone detection tools. *IEEE Transactions on Software Engineering* 33, 9 (2007), 577–591.

[14] Tegawendé F Bissyandé, Ferdian Thung, David Lo, Lingxiao Jiang, and Laurent Réveillere. 2013. Popularity, interoperability, and impact of programming languages in 100,000 open source projects. In *Proceedings of the 37th Annual Computer Software and Applications Conference*. IEEE, 303–312.

[15] TegawendÃľ F. BissyandÃľ. 2015. Harvesting Fix Hints in the History of Bugs. *arXiv:1507.05742 [cs]* (July 2015). arXiv: 1507.05742.

[16] R. Brixtel, M. Fontaine, B. Lesner, C. Bazin, and R. Robbes. 2010. Language-Independent Clone Detection Applied to Plagiarism Detection. In *2010 10th IEEE Working Conference on Source Code Analysis and Manipulation*. 77–86.

[17] Raymond P. L. Buse and Westley Weimer. 2012. Synthesizing API Usage Examples. In *Proceedings of the 34th International Conference on Software Engineering*. IEEE Press, Piscataway, NJ, USA, 782–792.

[18] J. R. Cordy and C. K. Roy. 2011. The NiCad Clone Detector. In *Proceedings of the 19th International Conference on Program Comprehension*. IEEE, 219–220.

[19] Creative Commons Attribution-ShareAlike 3.0 Unported License. 2016. https://creativecommons.org/licenses/by-sa/3.0/legalcode. (2016). last accessed 25.02.2017.

[20] Barthélémy Dagenais and Martin P Robillard. 2012. Recovering traceability links between an API and its learning resources. In *Proceedings of the 34th International Conference on Software Engineering*. IEEE, 47–57.

[21] Manuel Egele, Maverick Woo, Peter Chapman, and David Brumley. 2014. Blanket Execution: Dynamic Similarity Testing for Program Binaries and Components.. In *Usenix Security*. 303–317.

[22] T. Eisenbarth, R. Koschke, and D. Simon. 2003. Locating features in source code. *IEEE Transactions on Software Engineering* 29, 3 (March 2003), 210–224.

[23] FaCoY. 2017. https://github.com/facoy/facoy. (2017).

[24] G. W. Furnas, T. K. Landauer, L. M. Gomez, and S. T. Dumais. 1987. The Vocabulary Problem in Human-system Communication. *Commun. ACM* 30, 11 (Nov. 1987), 964–971.

[25] Q. Gao, H. Zhang, J. Wang, Y. Xiong, L. Zhang, and H. Mei. 2015. Fixing Recurring Crash Bugs via Analyzing Q&A Sites (T). In *2015 30th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. 307–318.

[26] Google Code Jam. 2017. https://code.google.com/codejam/. (Jan. 2017).

[27] C. Le Goues, T. Nguyen, S. Forrest, and W. Weimer. 2012. GenProg: A Generic Method for Automatic Software Repair. *IEEE Transactions on Software Engineering* 38, 1 (Jan. 2012), 54–72.

[28] N. GÃűde and R. Koschke. 2009. Incremental Clone Detection. In *2009 13th European Conference on Software Maintenance and Reengineering*. 219–228.

[29] Sonia Haiduc, Gabriele Bavota, Andrian Marcus, Rocco Oliveto, Andrea De Lucia, and Tim Menzies. 2013. Automatic Query Reformulations for Text Retrieval in Software Engineering. In *Proceedings of the 2013 International Conference on Software Engineering*. IEEE Press, Piscataway, NJ, USA, 842–851.

[30] Raphael Hoffmann, James Fogarty, and Daniel S Weld. 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*. ACM, 13–22.

[31] Reid Holmes and Gail C. Murphy. 2005. Using Structural Context to Recommend Source Code Examples. In *Proceedings of the 27th International Conference on Software Engineering*. ACM, New York, NY, USA, 117–125.

[32] R. Holmes, R. J. Walker, and G. C. Murphy. 2006. Approximate Structural Context Matching: An Approach to Recommend Relevant Examples. *IEEE Transactions on Software Engineering* 32, 12 (Dec. 2006), 952–970.

[33] Katsuro Inoue, Yusuke Sasaki, Pei Xia, and Yuki Manabe. 2012. Where Does This Code Come from and Where Does It Go? - Integrated Code History Tracker for Open Source Systems. In *Proceedings of the 34th International Conference on Software Engineering (ICSE '12)*. IEEE Press, Piscataway, NJ, USA, 331–341.

[34] Lingxiao Jiang, Ghassan Misherghi, Zhendong Su, and Stephane Glondu. 2007. Deckard: Scalable and accurate tree-based detection of code clones. In *Proceedings of the 29th international conference on Software Engineering*. IEEE Computer Society, 96–105.

[35] Lingxiao Jiang and Zhendong Su. 2009. Automatic mining of functionally equivalent code fragments via random testing. In *Proceedings of the eighteenth international symposium on Software testing and analysis*. ACM, 81–92.

[36] Elmar Juergens, Florian Deissenboeck, and Benjamin Hummel. 2010. Code similarities beyond copy & paste. In *Software Maintenance and Reengineering (CSMR), 2010 14th European Conference on*. IEEE, 78–87.

[37] RenÃľ Just, Darioush Jalali, and Michael D. Ernst. 2014. Defects4J: A Database of Existing Faults to Enable Controlled Testing Studies for Java Programs. In *Proceedings of the 2014 International Symposium on Software Testing and Analysis*. ACM, New York, NY, USA, 437–440.

[38] Eirini Kalliamvakou, Georgios Gousios, Kelly Blincoe, Leif Singer, Daniel M German, and Daniela Damian. 2014. The promises and perils of mining GitHub. In *Proceedings of the 11th working conference on mining software repositories*. ACM, 92–101.

[39] Toshihiro Kamiya, Shinji Kusumoto, and Katsuro Inoue. 2002. CCFinder: a multilinguistic token-based code clone detection system for large scale source code. *IEEE Transactions on Software Engineering* 28, 7 (2002), 654–670.

[40] Y. Ke, K. T. Stolee, C. L. Goues, and Y. Brun. 2015. Repairing Programs with Semantic Code Search (T). In *2015 30th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. 295–306.

[41] Yalin Ke, Kathryn T Stolee, Claire Le Goues, and Yuriy Brun. 2015. Repairing programs with semantic code search (t). In *Automated Software Engineering (ASE), 2015 30th IEEE/ACM International Conference on*. IEEE, 295–306.

[42] I. Keivanloo, J. Rilling, and P. Charland. 2011. SeClone - A Hybrid Approach to Internet-Scale Real-Time Code Clone Search. In *Proceedings of the 19th International Conference on Program Comprehension*. 223–224.

[43] Iman Keivanloo, Juergen Rilling, and Ying Zou. 2014. Spotting Working Code Examples. In *Proceedings of the 36th International Conference on Software Engineering*. ACM, New York, NY, USA, 664–675.

[44] D. Kim, J. Nam, J. Song, and S. Kim. 2013. Automatic patch generation learned from human-written patches. In *2013 35th International Conference on Software Engineering (ICSE)*. 802–811. DOI: http://dx.doi.org/10.1109/ICSE.2013.6606626

[45] H. Kim, Y. Jung, S. Kim, and K. Yi. 2011. MeCC: Memory comparison-based clone detector. In *Proceedings of the 33rd International Conference on Software Engineering*. IEEE, 301–310.

[46] A. J. Ko, B. A. Myers, M. J. Coblenz, and H. H. Aung. 2006. An Exploratory Study of How Developers Seek, Relate, and Collect Relevant Information during Software Maintenance Tasks. *IEEE Transactions on Software Engineering* 32, 12 (Dec. 2006), 971–987.

[47] Raghavan Komondoor and Susan Horwitz. 2001. Using Slicing to Identify Duplication in Source Code. In *Proceedings of the 8th International Symposium on Static Analysis*. Springer-Verlag, London, UK, UK, 40–56.

[48] Anil Koyuncu, TegawendÃľ F. BissyandÃľ, Dongsun Kim, Jacques Klein, Martin Monperrus, and Yves Le Traon. 2017. Impact of Tool Support in Patch Construction. In *Proceedings of the 26th ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA 2017)*. ACM, New York, NY, USA, 237–248.

[49] J. Krinke. 2001. Identifying similar code with program dependence graphs. In *Proceedings Eighth Working Conference on Reverse Engineering*. 301–309.

[50] D. E. Krutz and E. Shihab. 2013. CCCD: Concolic code clone detection. In *2013 20th Working Conference on Reverse Engineering (WCRE)*. 489–490.

[51] Frederick Wilfrid Lancaster and Emily Gallup Fayen. 1973. *Information Retrieval: On-line.* Melville Publishing Company.

[52] Mu-Woong Lee, Jong-Won Roh, Seung-won Hwang, and Sunghun Kim. 2010. Instant Code Clone Search. In *Proceedings of the 18th International Symposium on Foundations of Software Engineering (FSE '10)*. ACM, 167–176.

[53] OtÃąvio A. L. Lemos, Adriano C. de Paula, Felipe C. Zanichelli, and Cristina V. Lopes. 2014. Thesaurus-based Automatic Query Expansion for Interface-driven Code Search. In *Proceedings of the 11th Working Conference on Mining Software Repositories*. ACM, New York, NY, USA, 212–221.

[54] O. A. L. Lemos, A. C. de Paula, H. Sajnani, and C. V. Lopes. 2015. Can the use of types and query expansion help improve large-scale code search?. In *2015 IEEE 15th International Working Conference on Source Code Analysis and Manipulation (SCAM)*. 41–50.

[55] Sihan Li, Xusheng Xiao, Blake Bassett, Tao Xie, and Nikolai Tillmann. 2016. Measuring Code Behavioral Similarity for Programming and Software Engineering Education. In *Proceedings of the 38th International Conference on Software Engineering Companion*. ACM, New York, NY, USA, 501–510.

[56] Zhenmin Li, Shan Lu, Suvda Myagmar, and Yuanyuan Zhou. 2004. CP-Miner: a tool for finding copy-paste and related bugs in operating system code. In *Proceedings of the 6th conference on Symposium on Opearting Systems Design & Implementation - Volume 6*. USENIX Association, Berkeley, CA, USA, 20–20.

[57] Chao Liu, Chen Chen, Jiawei Han, and Philip S. Yu. 2006. GPLAG: Detection of Software Plagiarism by Program Dependence Graph Analysis. In *Proceedings of the 12th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*. ACM, New York, NY, USA, 872–881.

[58] Kui Liu, Dongsun Kim, TegawendâĹŽÂľ F. BissyandâĹŽÂľ, Shin Yoo, and Yves Le Traon. 2017. Mining Fix Patterns for FindBugs Violations. *arXiv:1712.03201 [cs]* (Dec. 2017).

[59] Fei Lv, Hongyu Zhang, Jian-guang Lou, Shaowei Wang, Dongmei Zhang, and Jianjun Zhao. 2015. CodeHow: Effective Code Search based on API Understanding and Extended Boolean Model. In *Proceedings of the 2015 30th IEEE/ACM International Conference on Automated Software Engineering (ASE '15)*. IEEE Computer Society, Lincoln, Nebraska, USA, 260–270.

[60] Lena Mamykina, Bella Manoim, Manas Mittal, George Hripcsak, and Björn Hartmann. 2011. Design lessons from the fastest q&a site in the west. In *Proceedings of the SIGCHI conference on Human factors in computing systems*. ACM, 2857–2866.

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

[62] A. Marcus and J. I. Maletic. 2001. Identification of high-level concept clones in source code. In *Proceedings 16th Annual International Conference on Automated Software Engineering (ASE 2001)*. 107–114.

[63] Lee Martie, AndrÃľ van der Hoek, and Thomas Kwak. 2017. Understanding the Impact of Support for Iteration on Code Search. In *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*. ACM, New York, NY, USA, 774–785.

[64] Michael McCandless, Erik Hatcher, and Otis Gospodnetic. 2010. *Lucene in Action, Second Edition: Covers Apache Lucene 3.0.* Manning Publications Co., Greenwich, CT, USA.

[65] C. McMillan, M. Grechanik, D. Poshyvanyk, C. Fu, and Q. Xie. 2012. Exemplar: A Source Code Search Engine for Finding Highly Relevant Applications. *IEEE Transactions on Software Engineering* 38, 5 (Sept. 2012), 1069–1087.

[66] Collin McMillan, Mark Grechanik, Denys Poshyvanyk, Qing Xie, and Chen Fu. 2011. Portfolio: finding relevant functions and their usage. In *Proceeding of the 33rd international conference on Software engineering*. ACM, Waikiki, Honolulu, HI, USA, 111–120. ACM ID: 1985809.

[67] Laura Moreno, Gabriele Bavota, Massimiliano Di Penta, Rocco Oliveto, and Andrian Marcus. 2015. How Can I Use This Method?. In *Proceedings of the 37th International Conference on Software Engineering - Volume 1*. IEEE Press, Piscataway, NJ, USA, 880–890.

[68] Seyed Mehdi Nasehi, Jonathan Sillito, Frank Maurer, and Chris Burns. 2012. What makes a good code example?: A study of programming Q&A in StackOverflow. In *Software Maintenance (ICSM), 2012 28th IEEE International Conference on*. IEEE, 25–34.

[69] Haoran Niu, Iman Keivanloo, and Ying Zou. 2016. Learning to rank code examples for code search engines. *Empirical Software Engineering* (Jan. 2016), 1–33.

[70] Praveen Pathak, Michael Gordon, and Weiguo Fan. 2000. Effective information retrieval using genetic algorithms based matching functions adaptation. In *System Sciences, 2000. Proceedings of the 33rd Annual Hawaii International Conference on*. IEEE, 8–pp.

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

[72] D. Poshyvanyk, Y. G. Gueheneuc, A. Marcus, G. Antoniol, and V. Rajlich. 2007. Feature Location Using Probabilistic Ranking of Methods Based on Execution Scenarios and Information Retrieval. *IEEE Transactions on Software Engineering* 33, 6 (June 2007), 420–432.

[73] Chanchal K Roy, James R Cordy, and Rainer Koschke. 2009. Comparison and evaluation of code clone detection techniques and tools: A qualitative approach. *Science of Computer Programming* 74, 7 (2009), 470–495.

[74] Hitesh Sajnani, Vaibhav Saini, Jeffrey Svajlenko, Chanchal K Roy, and Cristina V Lopes. 2016. SourcererCC: scaling code clone detection to big-code. In *Proceedings of the 38th International Conference on Software Engineering*. ACM, 1157–1168.

[75] Gerard Salton and Michael J. McGill. 1986. *Introduction to Modern Information Retrieval.* McGraw-Hill, Inc., New York, NY, USA.

[76] G. Salton, A. Wong, and C. S. Yang. 1975. A Vector Space Model for Automatic Indexing. *Commun. ACM* 18, 11 (Nov. 1975), 613–620.

[77] Huascar Sanchez. 2013. SNIPR: Complementing Code Search with Code Retargeting Capabilities. In *Proceedings of the 2013 International Conference on Software Engineering (ICSE '13)*. IEEE Press, Piscataway, NJ, USA, 1423–1426.

[78] Niko Schwarz, Mircea Lungu, and Romain Robbes. 2012. On How Often Code is Cloned Across Repositories. In *Proceedings of the 34th International Conference on Software Engineering*. IEEE Press, Piscataway, NJ, USA, 1289–1292.

[79] Raphael Sirres, Tegawendé F. Bissyandé, Dongsun Kim, David Lo, Jacques Klein, Kisub Kim, and Yves Le Traon. 2017. Augmenting and Structuring User Queries to Support Efficient Free-Form Code Search. *Empirical Software Engineering* (2017), (to appear).

[80] Kathryn T. Stolee, Sebastian Elbaum, and Daniel Dobos. 2014. Solving the Search for Source Code. *ACM Trans. Softw. Eng. Methodol.* 23, 3 (June 2014), 26:1–26:45. DOI: http://dx.doi.org/10.1145/2581377

[81] Fang-Hsiang Su, Jonathan Bell, Kenneth Harvey, Simha Sethumadhavan, Gail Kaiser, and Tony Jebara. 2016. Code Relatives: Detecting Similarly Behaving Software. In *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering (FSE 2016)*. ACM, 702–714.

[82] Siddharth Subramanian, Laura Inozemtseva, and Reid Holmes. 2014. Live API documentation. In *Proceedings of the 36th International Conference on Software Engineering*. ACM, 643–652.

[83] Jeffrey Svajlenko, Judith F Islam, Iman Keivanloo, Chanchal K Roy, and Mohammad Mamun Mia. 2014. Towards a big data curated benchmark of inter-project code clones. In *Software Maintenance and Evolution (ICSME), 2014 IEEE International Conference on*. IEEE, 476–480.

[84] Jeffrey Svajlenko and Chanchal K Roy. 2015. Evaluating clone detection tools with bigclonebench. In *Software Maintenance and Evolution (ICSME), 2015 IEEE International Conference on*. IEEE, 131–140.

[85] Christoph Treude and Martin P Robillard. 2016. Augmenting API documentation with insights from Stack Overflow. In *Proceedings of the 38th International Conference on Software Engineering*. ACM, 392–403.

[86] Martin White, Michele Tufano, Christopher Vendome, and Denys Poshyvanyk. 2016. Deep Learning Code Fragments for Code Clone Detection. In *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering*. ACM, New York, NY, USA, 87–98.

[87] Xin Xia, Lingfeng Bao, David Lo, Pavneet Singh Kochhar, Ahmed E. Hassan, and Zhenchang Xing. 2017. What do developers search for on the web? *Empirical Software Engineering* (April 2017), 1–37.

[88] Tao Xie and Jian Pei. 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, NY, USA, 54–57.

[89] Le Zhao and Jamie Callan. 2010. Term Necessity Prediction. In *Proceedings of the 19th ACM International Conference on Information and Knowledge Management*. ACM, New York, NY, USA, 259–268.

[90] Le Zhao and Jamie Callan. 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*. ACM, New York, NY, USA, 515–524.