CodeMatcher: Searching Code Based on Sequential Semantics of Important Query Words

CHAO LIU, Zhejiang University, China
XIN XIA, Huawei, China
DAVID LO, Singapore Management University, Singapore
ZHIWE LIU, Baidu Inc., China
AHMED E. HASSAN, Queen's University, Canada
SHANPING LI, Zhejiang University, China

To accelerate software development, developers frequently search and reuse existing code snippets from a large-scale codebase, e.g., GitHub. Over the years, researchers proposed many information retrieval (IR)-based models for code search, but they fail to connect the semantic gap between query and code. An early successful deep learning (DL)-based model DeepCS solved this issue by learning the relationship between pairs of code methods and corresponding natural language descriptions. Two major advantages of DeepCS are the capability of understanding irrelevant/noisy keywords and capturing sequential relationships between words in query and code. In this article, we proposed an IR-based model CodeMatcher that inherits the advantages of DeepCS (i.e., the capability of understanding the sequential semantics in important query words), while it can leverage the indexing technique in the IR-based model to accelerate the search response time substantially. CodeMatcher first collects metadata for query words to identify irrelevant/noisy ones, then iteratively performs fuzzy search with important query words on the codebase that is indexed by the Elasticsearch tool and finally reranks a set of returned candidate code according to how the tokens in the candidate code snippet sequentially matched the important words in a query. We verified its effectiveness on a large-scale codebase with ~41K repositories. Experimental results showed that CodeMatcher achieves an MRR (a widely used accuracy measure for code search) of 0.60, outperforming DeepCS, CodeHow, and UNIF by 82%, 62%, and 46%, respectively. Our proposed model is over 1.2K times faster than DeepCS. Moreover, CodeMatcher outperforms two existing online search engines (GitHub and Google search) by 46% and 33%, respectively, in terms of MRR. We also observed that: fusing the advantages of IR-based and DL-based models is promising; improving the quality of method naming helps code search, since method name plays an important role in connecting query and code.

CCS Concepts: • **Software and its engineering → Search-based software engineering;**

Additional Key Words and Phrases: Code search, code indexing, mining software repositories, information retrieval

Authors' addresses: C. Liu and S. Li, Zhejiang University, Lingyin Street, Hangzhou, Zhejiang, 310013, China; emails: {liuchaoo, shan}@zju.edu.cn; X. Xia (corresponding author), Huawei, Xixing Street, Hangzhou, Zhejiang, 310051, China; email: xin.xia@acm.org; D. Lo, Singapore Management University, 80 Stamford Road, 178902, Singapore; email: davidlo@smu.edu.sg; Z. Liu, Baidu Inc., 701 Naxian Road, Shanghai, 201210, China; email: zhiweiliu03@baidu.com; A. E. Hassan, Queen's University, 133 Princess Street, Kingston, K7L 1A8, Ontario, Canada; email: ahmed@queensu.ca.

1049-331X/2021/09-ART12 $15.00
https://doi.org/10.1145/3465403

ACM Reference format:
Chao Liu, Xin Xia, David Lo, Zhiwe Liu, Ahmed E. Hassan, and Shanping Li. 2021. CodeMatcher: Searching Code Based on Sequential Semantics of Important Query Words. *ACM Trans. Softw. Eng. Methodol.* 31, 1, Article 12 (September 2021), 37 pages.
https://doi.org/10.1145/3465403

1 INTRODUCTION

Code search is the most frequent activity in software development [46, 61, 63, 65], as developers favor searching existing code and learning from them just-in-time when meeting a programming issue [5, 6]. Reusing existing diverse code from millions of open-source repositories (e.g., in GitHub [26]) can maximize developers' productivity [15, 35, 45, 64, 66]. During software development, it was observed that more than 90% of developers' code search efforts are used to find code snippet [3], thus this study focuses on searching code methods following previous studies [2, 18, 41] instead of finding relevant repositories (e.g., from GitHub) [17, 60]. Moreover, different from the studies of searching/synthesizing API usage examples [7, 23], our study focus on the query with natural language description (e.g., "how to convert InputStream to String") instead of the query with the expected API (class) names (e.g., "InputStream" or "StringBuilder"); the objective of our study is to directly find relevant code methods from codebase with similar semantics as the query, not to filter/synthesize representative API usage examples based on a set of candidate methods that were previously searched from codebase using the inputted API name.

Prior works on code search studies start with leveraging **information retrieval (IR)** techniques (e.g., Koders [3] and Krugle [30]), which regard method code as text and match keywords in query with indexed methods [2, 30]. But their performance is poor due to three reasons: *(1) short and diverse queries,* keywords matching can hardly represent various search requirements due to insufficient context; *(2) representation of method as text,* method has a structure that carries specialized semantics [3]; *(3) poor ranking/sorting,* existing ranking approaches cannot precisely measure the semantic relevancy between query and candidate code [23, 28]. To address these issues, many past studies focus on query expansion and reformulation [20, 21, 39, 45, 53]. For example, the Sourcerer model [2] extended the textual content of a method with structural information. The model proposed by Lu et al. [39] expanded a query with synonyms generated from WordNet [49] and matched keywords to method signatures. The CodeHow model [41] extended query with related APIs and searched code methods with matched APIs and query keywords through an extended Boolean model. To improve the IR-based retrieval, Keivanloo et al. [28] clustered code methods based on the mined fine-grained code patterns. For each cluster, they selected one representative method to exclude duplicated methods with highly syntactic similarity. Afterward, they reranked the selected representative methods from all clusters in terms of code features (e.g., conciseness and completeness).

Recently, Gu et al. [18] observed that IR-based models have two problems: *(1) semantic gap,* keywords cannot adequately represent high-level intent implied in queries, and they also cannot reflect low-level implementation details in code; *(2) representation gap,* query and code are semantically related, but they may not share common lexical tokens, synonyms, or language structures. To connect the semantic gap between query and code, early IR-based models mainly focused on expanding queries with related API (class) names [41, 72]. To further solve the issue of representation gap, Gu et al. [18] proposed a **deep learning (DL)**-based model named DeepCS. It jointly embeds method code and natural language description into a high-dimensional vector space, where the methods with high similarities to a query are retrieved. Their experiments on a large-scale codebase collected from GitHub verified the model validity, showing substantial advantages over two representative IR-based models, Sourcerer [2] and CodeHow [41].

Gu et al. [18] attributed the success of DeepCS, over IR-based models, to the capability of handling irrelevant/noisy keywords in queries and recognizing semantically related words between queries and code methods. However, DeepCS involves a complicated optimization process with a lot of manually set parameters. It requires more than 50 hours for model training on a machine with an Nvidia K40 GPU, and developers need to wait for about 6 minutes on a code search. Thus, it is necessary to explore the possibility to simplify DeepCS in a controllable way for practical usages. To address this challenge, we proposed an IR-based model CodeMatcher that maintains the advantageous features in DeepCS as described above. The IR technique is adopted because it can avoid time-consuming training and improve code search efficiency, and researchers do not need to take a lot of efforts for manually tuning complex parameters in DL-based model. Generally, CodeMatcher leverages Elasticsearch [16], a Lucene-based text search engine, to index codebase and perform fuzzy search with the identified important keywords from search queries. To improve query understanding, CodeMatcher removes noisy keywords according to some collected metadata and replaces irrelevant keywords with appropriate synonyms extracted from the codebase. To optimize the ranking of code methods searched by Elasticsearch, we designed two strategies (S_{name} and S_{body}) to rerank the code methods. For a pair of query and method, S_{name} measures the semantic similarity between the query and method name. Meanwhile, S_{body} measures the semantic similarity between the query and method body (i.e., the method implementation part). Like DeepCS, the similarity measurements consider the semantically related important words between queries and code methods and their sequential relationships. Our study investigated the following **research questions (RQs)**:

RQ1: *Can CodeMatcher outperform the baseline models?* We tested CodeMatcher and the models DeepCS [18], CodeHow [41], and UNIF [8] on a large-scale testing data with ~41K repositories. Experimental results showed that CodeMatcher achieves an MRR of 0.60, substantially outperforming DeepCS, CodeHow, and UNIF by 82%, 62%, and 46%, respectively.

RQ2: *Is CodeMatcher faster than the baseline models?* CodeMatcher needs no model training like DeepCS and UNIF, and it only takes 0.3 s for code search per query, which is over 1.2K and 1.5K times faster than two DL-based models DeepCS and UNIF, respectively. Besides, CodeMatcher works eight times faster than the IR-based model CodeHow.

RQ3: *How do the CodeMatcher components contribute to the code search performance?* The CodeMatcher consists of three components, namely, query understanding, fuzzy search, and reranking. The query understanding mainly collects metadata for the other two components. When removing the reranking component, the search performance is reduced by 22% in terms of MRR (from 0.60 to 0.47). However, the model with only fuzzy search component still outperforms baseline models DeepCS, CodeHow, and UNIF by 42%, 27%, and 15%, respectively. Therefore, all the components are necessarily required for CodeMatcher.

RQ4: *Can CodeMatcher outperform existing online code search engines?* The proposed model CodeMatcher outperforms the popular online code search engines, GitHub search and Google search, by 46% and 33% in terms of MRR, respectively. Moreover, we also found that combining Google search to CodeMatcher can further improve the MRR of CodeMatcher from 0.60 to 0.64. These results imply the merit of CodeMatcher in practical usage.

We offer these results to the software analytics community and suggest that sophisticated techniques like DL worth a try for innovation, but their computation efficiency should also be considered. Meanwhile, although the IR-based model performs better than DL-based models in terms of accuracy and time-efficiency in our experiment, the IR-based model cannot address queries with complex syntax and tolerate errors as the DL-based model. Thus, we recommend fusing the advantages of IR-based and DL-based models in the future. Besides, the high performance of

CodeMatcher indicates that the method name is significant for code search, because it usually shares similar syntactic and semantic format as a search query. Thus, improving the quality of developers' method naming also helps code search.

The main contributions of this study are:

- We propose an IR-based model, CodeMatcher, that not also runs fast but also inherits the advantages of the DL-based model DeepCS. It can better understand query semantics by addressing irrelevant/noisy words and correctly map the query-code semantics by capturing the sequential relationship between important words.
- We evaluate the effectiveness of CodeMatcher on a large-scale codebase collected by us from GitHub. The experimental results show that CodeMatcher substantially outperforms three existing models (DeepCS, CodeHow, and UNIF).
- We compare CodeMatcher with two existing online search engines, GitHub and Google search. CodeMatcher also shows substantial advantages over these two popular online search engines in code search.

The remainder of this article is organized as follows: Section 2 describes the background of the IR-based and DL-based models for code search. Section 3 presents the proposed model CodeMatcher and shows how CodeMatcher is simplified from the DL-based model DeepCS. Sections 4 and 5 present the experiment setup and results, respectively, followed by the discussion in Section 6 and the implication in Section 7. Section 9 describes related work, and Section 8 presents the threats to validation and model limitation. Section 10 summarizes this study and presents future work.

2 BACKGROUND

As illustrated in Table 1, for a search query (e.g., "how to read a text line by line in java"), code search models aim to find relevant code methods (e.g., "readTextLineByLine()") from a codebase with many candidate code methods. How to correctly map the semantics between the search query and code methods is the key to build a code search model. This section briefly introduces the background of two code search categories: IR-based and DL-based models.

2.1 IR-based Code Search

Traditional code search models (e.g., CodeHow [41]) mainly depend on IR techniques. Generally, the IR-based model builds an index for terms (i.e., each word token) in code methods, so the keywords generated from search query can match the expected code methods quickly by checking the index. Afterward, the model recommends the top-n relevant code methods according to a designed keywords-terms matching degree. During the indexing, the method name and body are commonly regarded as two different components for a method [41]. This is because method name (e.g., "readTextLineByLine()") is often defined in natural language whose semantic representation is close to the query (e.g., "how to read a text line by line"). But the method body implements the goal of the method name in programming languages (e.g., "new Buffered Reader()" and "br.close()"). Therefore, when matching keywords for a code method, a weighted sum of matching degrees on two different method components is utilized [41]. Due to the success of existing search engines (e.g., Elasticsearch [16]), code search researchers can easily build an IR-based model using the APIs provided by a search engine. The researchers can focus more on query understanding to address the semantic gap between query and code method. Namely, the IR-based models take more effort on how to generate correct keywords and how to calculate the keywords-terms matching degree. As illustrated in Table 1, two code methods may contain many keywords related to the search

Table 1. Example of Code Search with a Natural Language Query

Query: how to read a text line by line in java

An irrelevant code:

```
public String HowToRead(String path){
   FileReader fr = new FileReader(path);
   BufferedReader br = new BufferedReader(fr);
   String firstline = br.readLine();
   return firstline;
}
```

A relevant code:

```
public void readTextLineByLine(String file) {
   BufferedReader br = new BufferedReader(new FileInputStream(file));
   String line = null;
   while ((line = br.readLine())!= null) {
      System.out.println(line);
   }
   br.close();
}
```

query, but how to identify the relevant code and exclude the irrelevant one is still a challenging issue. CodeHow [41] is an IR-based model. It leverages an Extended boolean model [62] to measure the degree to which query keywords match a method name and body, respectively, and sort code methods based on a weighted sum of matching degrees for method name and body. To spot representative methods in the searched candidate methods, some models cluster candidates based on code patterns [28, 50, 71]. For example, Keivanloo et al. [28] represented a method by an encoded pattern in terms of a set of code entities (e.g., method blocks). After retrieving a list of candidate methods for a query, they transformed them into vectors within a latent space where the methods with similar code patterns can be easily clustered. Afterward, they reranked representative methods in clusters based on their features (e.g., code conciseness and completeness).

2.2 DL-based Code Search

Gu et al. [18] indicated that existing IR-based models (e.g., CodeHow) do not work well because they have difficulties in query understanding. They cannot effectively address irrelevant/noisy keywords in the queries, so the IR-based models cannot find code methods that highly related to a query [18]. To overcome this challenge, Gu et al. [18] provided a model DeepCS that leverages the DL technique to better understand query semantics and correctly search related code from a large scale of the codebase. Generally, to map the semantics of a search query and code methods, DeepCS aims to leverage DL the technique to vectorize queries and methods at first and learn their semantic relationships directly. In this way, the methods relevant to a query can be easily identified by calculating the cosine similarities. In specific, the DL-based model DeepCS [18] encoded a pair of query and method into vectors with a fixed-size of vocabulary and input the vectorized query-method pairs into different **long-short term memory networks (LSTMs)** [22]. The LSTM model aims to capture the sequential relationship between words in query or method. The parameters in the networks are optimized by a ranking loss function, which rewards the related query-method pairs and penalizes the unrelated ones. As the ground-truth query-method pair is not easy to obtain, DL-based models used the first line of method comment to represent the corresponding

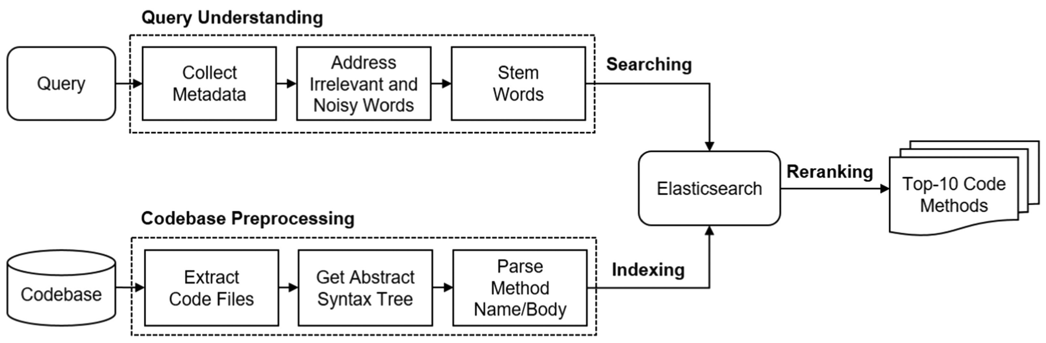

Fig. 1. Overall framework of CodeMatcher.

query. Similarly, the unrelated query to a method is the comment randomly selected from other commented methods. More details on DeepCS can be found in Reference [18].

3 CODEMATCHER

This section first presents the motivation of building a simplified model for DeepCS [18], and then describes the implementation details of the proposed model CodeMatcher.

3.1 Motivation

Different from the existing IR-based models, the DL-based model DeepCS can: (1) understand irrelevant/noisy keywords by word embedding technique, including synonyms and the words that have never appeared in the code methods; (2) capture sequential relationships between words in query and code methods via the LSTM model; and (3) map query intent to the semantics of code methods by measuring their semantic similarity [18]. However, due to the high complexity of DeepCS, it takes hours for model training and the code search is also time-consuming for practical usage. Therefore, it is necessary to find a way to simplify the model complexity but retain the beneficial features of the DL-based model. In this study, we simplify the DL-based model DeepCS into an IR-based model called CodeMatcher that maintains the advantages of DeepCS. Generally, we remove noisy keywords according to some collected metadata; replace irrelevant keywords with appropriate synonyms extracted from codebase; design a code reranking approach by measuring the semantic similarity between query and code, which also considers the sequential relationship between words in query/code. The following subsection presents the implementation details of CodeMatcher.

3.2 Implementation

The proposed model CodeMatcher performs code search in two phases following existing IR-based models, namely, code indexing and code search, as described in Section 2.1. Figure 1 illustrates the overall framework of CodeMatcher.

Phase-I: Codebase Preprocessing and Indexing. The first phase leverages the Elasticsearch [16], a Lucene-based text search engine for code indexing. Similar to existing IR-based models, we represent a code method as two components (i.e., method name and method body) and create an index for the method by calling the APIs provided by Elasticsearch. To extract method components from source code files, we developed a tool JAnalyzer,[1] which transforms a method into

[1]https://github.com/liuchaoss/janalyzer.

an **abstract syntax tree (AST)** with the Javaparser[2] [24] library and extracts method name (e.g., "convertInputStreamToString()" in Table 1) and body by traversing the AST. As JAnalyzer depends on the Javaparser tool, thus it can only support the Java language instead of other programming languages. Similar to the IR-based model CodeHow [41], the method body is parsed as a sequence of fully qualified tokens (e.g., Java.lang.String or System.io.File.readlines()) in method parameters, method body, and returned parameter.

Phase-II: Code Search. CodeMatcher takes three steps to perform code search on the indexed code methods for a query. The first step extracts some metadata for query understanding and addresses noisy and irrelevant query words. The second step quickly finds a pool of highly related code methods from the codebase according to the collected metadata for query words. The last step reranks the searched methods in the second step according to the matching degrees between the query and candidate methods.

Step-1: Query understanding. To address noisy words in a query, we removed the words that are not commonly used for programming. Specifically, we first identified word property (e.g., verb or noun) by using the Stanford Parser[3] [10, 73]. Then, we filtered out the question words and related auxiliary verb (e.g., how do) and excluded the verb-object/adpositional phrase on the programming language (e.g., in Java), as it is only used for the identification of programming language, while our study only focuses on Java projects as Gu et al. [18]. Afterward, we removed the words that are not the verbs, nouns, adjectives, adverbs, prepositions, or conjunctions, since the removed words (e.g., "a," "the," and non-English symbols) are rarely used for coding [1]. Moreover, to identify the irrelevant words in a query, we counted the frequency of each word occurring in the method names of the codebase. We replaced the irrelevant words (i.e., the words not appeared in codebase) by their synonyms generated by the WordNet [49]. If the WordNet generates multiple synonyms for a word, then we choose the synonym with the highest frequency in the codebase. Subsequently, we stemmed[4] the rest of query words to improve their generalizability in code search.

Moreover, except for the words property (e.g., verb or noun) and frequency, we identified the third metadata named "importance" for query words before performing code search. The word importance refers to how often a query word is used for code programming. It can help CodeMatcher identify related important keywords between query and code methods. The importance is categorized by five levels, as shown in Table 2. For example, the most important one (i.e., level 5) is the JDK noun, e.g., "InputStream" or "readLine()." These metadata can help query understanding and forming keywords for code search. Verbs and nouns are important (level 4), as they can represent most of the semantics in code methods. This is because method/API names are typically made of verbs or verb phrases (such as "stop" or "sendMessage") while variable names are often written in nouns (e.g., "student" or "processor"), as described in the Google Java Style Guide.[5] We assigned JDK nouns with higher importance (level 5), because they are unique (namely, it cannot be replaced by synonyms) and important to represent the semantics in code methods. For example, the JDK nouns "Inputstream" and "String" are important for the code with the semantics "convert Inputstream to String." Adjectives and adverbs are assigned lower importance (level 3), because they cannot indicate precise meaning without their corresponding noun/verb. Next, we assigned preposition/conjunction with the importance level 2. Finally, we assigned the other (often meaningless) symbols the lowest importance level (level 1).

[2]https://github.com/javaparser/javaparser.
[3]https://nlp.stanford.edu/software/lex-parser.shtml.
[4]https://pythonprogramming.net/stemming-nltk-tutorial/.
[5]https://google.github.io/styleguide/javaguide.html.

Table 2. Five Word Importance Levels for Programming Based on the Word Property (e.g., Verb or Noun) and Whether the Word Is a Class Name in JDK

Level	Condition	Examples
5	JDK Noun	"Inputstream" and "readLine()"
4	Verb or Non-JDK Noun	"convert" and "whitespace"
3	Adjective or Adverb	"numeric" and "decimal"
2	Preposition or Conjunction	"from" and "or"
1	Other	Number and Non-English Symbols

Step-2: Iterative fuzzy search. To map the semantics of the preprocessed query and indexed code methods, we utilized the filtered query words in Step-1 to generate keywords for code search. Here, we only match the keywords with indexed method names to quickly narrow the search space. To perform code search, we launched an iterative fuzzy match on indexed method names with a regular string.[6] The string is formed by all remaining query words in order as "$. * word_1 . * \cdots . * word_n . *$." We used the regular match instead of the term match to capture the sequential relationship between words as DeepCS [18]. For example, "convert int to string" should be different from "convert string to int." Afterward, if the total number of returned methods is no more than 10, then we removed the least important word with lower frequency one at a time according to their metadata and performed the fuzzy search again until only no query word left. For each search round, we filtered out the repeated method by comparing the MD5 hash[7] values of their source code.

Step-3: Reranking. To refine the method rankings returned from Step-2, we designed a metric (S_{name}) to measure the matching degree between query and method name as Equation (1). A larger value of S_{name} indicates a higher-ranked method with more overlapped tokens between query and method name in order. Moreover, if equal S_{name} exists, then we boosted the rank of a method with a higher matching score (S_{body}) between query and method body, as calculated in Equation (2). Similar to S_{name}, a higher S_{body} value implies better token matching between query and tokens in method body orderly. However, different from S_{name}, we added the last term in S_{body} to represent the ratio of JDK APIs in method, in terms of the fully qualified tokens, because developers favor a method with more basic APIs (e.g., JDK) [42, 61]. After the above two rounds of ranking refinement, we returned the top-10 methods in the list.

Note that, although our tool JAnalyzer can extract Javadocs and comments for code, the CodeMatcher did not consider them when reranking the searched code following all the related studies [8, 12, 18, 25, 41]. This is because the code search task aims to solve the semantic gap between the query in natural language and the code in programming language. Besides, whether the comments and Javadocs written in natural language are helpful for code search is highly dependent on their quality and number. However, in practical usage, we cannot assume that any code always contains high-quality comment or Javadoc.

$$S_{name} = \frac{\#query\ words\ as\ keywords}{\#query\ words} \times \frac{\#characters\ in\ name\ orderly\ matched\ keywords}{\#characters\ in\ name} \tag{1}$$

[6]https://docs.python.org/3/library/re.html.
[7]https://docs.python.org/2/library/hashlib.html.

$$S_{body} = \frac{\#API\ words\ matched\ query\ words}{\#query\ words} \times \frac{Max[\#API\ words\ orderly\ matched\ query\ words]}{\#query\ words} \times \frac{\#JDKAPIs}{\#APIs} \tag{2}$$

Example. Table 3 illustrated an example for the first query *"convert an inputstream to a string"* in Table 5. From the token metadata, we can notice that both "inputstream" and "string" have level-5 importance (i.e., they are JDK objects), and they are frequently used for method naming (frequency > 3,442). With this metadata, CodeMatcher successively generates four candidate regular match strings on indexed method names. For the two returned methods, the first one ranked higher due to its larger matching scores on method name (S_{name}) and body (S_{body}).

4 EXPERIMENT SETUP

This section describes the investigated research questions, the collected large-scale dataset for model validation, and the widely used model evaluation criteria.

4.1 Research Questions

To verify the validity of the proposed model, this study investigates the following RQs:

RQ1. Can CodeMatcher outperform the baseline models?

The proposed model CodeMatcher aims to leverage IR technique to simplify the complexity of DeepCS while retaining its advantages, namely, its capability of addressing irrelevant/noisy keywords in queries and recognizing semantically related words between queries and code methods as described by Gu et al. [18]. To verify the validity of the proposed model, this RQ investigates whether the CodeMatcher outperforms the relevant models, including two DL-based models DeepCS [18] and UNIF [8], and an IR-based model, CodeHow [41].

RQ2. Is CodeMatcher faster than the baseline models?

We observed that training and testing the DL-based model DeepCS is time-consuming due to its high complexity. Therefore, we intend to analyze whether the simplified model CodeMatcher works faster than DeepCS substantially. Besides, we also compare the time-efficiency of CodeMatcher with UNIF [8] and CodeHow [41].

RQ3. How do the CodeMatcher components contribute to the code search performance?

CodeMatcher consists of three important components: the fuzzy search component retrieves an initial set of candidate relevant code from the indexed codebase, the reranking component sorts the candidate list based on the designed strategy, and the query understanding component collects metadata for the previous two components. They largely determine the performance of the code search. Thus, this RQ aims to analyze how much these components contribute to the model performance. The result can also help analyze the necessity of these components.

Table 3. An Example for CodeMatcher

Query: convert an inputstream to a string

(0) Indexed Codebase

Source Code 1:

```
public String convertInputStreamToString(InputStream is){
    InputStreamReader isr = new InputStreamReader(is);
    BufferedReader r = new BufferedReader(isr);
StringBuilder sb = new StringBuilder();
    String line;
    while ((line = r.readLine()) != null){
        sb.append(line);
    }
    return sb.toString();
}
```

Method Name: **convertInputStreamToString**

API Sequence: java.io.**InputStream**, java.io.**InputStream**Reader, java.lang.**String**Builder, java.lang.**String**, java.lang.**String**Builder.readline(), java.lang.**String**Builder.append(), java.lang.**String**Builder.**toString**(), java.io.**String**

Source Code 2:

```
public String convertInputStream2String(InputStream is){
    return convert(is);
}
```

Method Name: **convertInputStream**2**String**

API Sequence: java.io.**InputStream**, Util.**convert**(), java.lang.**String**

(1) Token Metadata

Token	convert	an	inputstream	to	a	string
Property	verb	other	noun	prep	other	noun
Frequency	39,292	0	3,442	22	0	52,369
Importance	4	1	5	2	1	5

(2) Keywords for Code Search

Regular Match String 1: .*convert.*inputstream.*to.*string.*

Regular Match String 2: .*convert.*inputstream.*string.*

Regular Match String 3: .*inputStream.*string.*

Regular Match String 4: .*String.*

(3) Reranking

Rank = 1, **convertInputStreamToString(){...}**

$S_{name} = \frac{4}{6} \times \frac{26}{26} = 0.67$, $S_{body} = \frac{3}{6} \times \frac{3}{6} \times \frac{8}{8} = 0.25$

Rank = 2, **convertInputStream2String(){...}**

$S_{name} = \frac{3}{6} \times \frac{24}{25} = 0.48$, $S_{body} = \frac{3}{6} \times \frac{2}{6} \times \frac{2}{3} = 0.11$

RQ4. Can CodeMatcher outperform existing online code search engines?

GitHub search[8] and Google search[9] are two commonly used search engines for developers to find code in practice. To measure the performance of GitHub/Google search, all the search

[8] https://github.com/search.

[9] https://google.com.

Table 4. Statistics of the Constructed Codebase

#Project	#Method	#Javadoc	#LOC
41,025	16,611,025	3,639,794	70,332,245

engines are tested with the same queries as CodeMatcher and they are set to search only GitHub repositories; for Google search, we use the following advanced settings: "site:github.com" and "file-type:java."

4.2 Dataset

Codebase. Originally, Gu et al. [18] collected 9,950 Java projects that have at least 20 stars in GitHub as the testing data of DeepCS. However, we cannot use their testing data to verify our IR-based model CodeMatcher, because Gu et al. [18] just provided a preprocessed data that can be used for DeepCS only. Therefore, we built a new and larger scale testing data for model evaluation. We crawled 41,025 Java repositories from GitHub created from July 2016 to December 2018 with more than five stars. The number of stars filter (>5 stars), which is different from Gu et al.'s [18] setting (i.e., at least 20 stars) is used so the testing data includes more Java projects. Besides, the time duration (July 2016 to December 2018) of our new testing data can ensure the non-overlapping with DeepCS' training data (created from August 2008 to June 2016). Table 4 shows that the new codebase contains ~17 million methods, and 21.91% of them have Javadoc comments that describe the corresponding methods.

Queries. To simulate a real-world code search scenario, we validated a code search model with three query sets in total of 174 queries as listed in Tables 5–7:

- *Queries*$_{50}$. The first query set was manually collected by Gu et al. [18] from Stack Overflow in a systematic way. These queries are top-50 voted Java programming questions[10] following three criteria: *(1) concrete,* a question should be a specific programming task, such as "How can I concatenate two arrays in Java?"; *(2) well-answered,* the accepted answers corresponding to the question should contain at least one code snippet; *(3) non-duplicated,* the question is not a duplicate of another question in the same collection.
- *Queries*$_{99}$. The second query set was collected from the CodeSearchNet challenge[11] that was built by Husain et al. [25]. The contained 99 queries[12] were common search queries from Bing that have high clickthrough rates to code with clearly technical keywords [25].
- *Queries*$_{25}$. The third query set has 25 queries based on the studies of Mishne et al. [50] and Keivanloo et al. [28]. Note that these two studies used API names as code search input. However, our study focused on the queries written in natural language. Therefore, we used the natural language descriptions provided in these two studies [28, 50] as our model inputs.

4.3 Baseline Models and Replication Package

In the experiment, three models are selected as our baseline models, including Gu et al.'s [18] DeepCS, Lv et al.'s [41] CodeHow, and Cambronero et al.'s UNIF [8]. To test baseline models on our codebase, we first preprocessed Java code files within all projects by our tool JAnalyzer.[13] It

[10] https://stackoverflow.com/questions/tagged/java?sort=votes.
[11] https://github.com/github/CodeSearchNet.
[12] https://github.com/github/CodeSearchNet/blob/master/resources/queries.csv.
[13] https://github.com/liuchaoss/janalyzer.

Table 5. The Queries$_{50}$ Collected from Gu et al. [18] and the Evaluation Results

No.	Query	DCS	CM	CH	UNIF
1	convert an inputstream to a string	2	1	2	2
2	create arraylist from array	8	1	1	4
3	iterate through a hashmap	1	1	1	NF
4	generating random integers in a specific range	1	1	5	4
5	converting string to int in java	8	1	2	1
6	initialization of an array in one line	1	1	1	2
7	how can I test if an array contains a certain value	NF	1	1	9
8	lookup enum by string value	NF	1	1	NF
9	breaking out of nested loops in java	NF	3	NF	NF
10	how to declare an array	1	1	1	NF
11	how to generate a random alpha-numeric string	1	1	NF	5
12	what is the simplest way to print a java array	1	1	NF	5
13	sort a map by values	5	1	1	1
14	fastest way to determine if an integer's square root is an integer	NF	NF	NF	NF
15	how can I concatenate two arrays in java	9	1	NF	3
16	how do I create a java string from the contents of a file	3	NF	5	2
17	how can I convert a stack trace to a string	2	1	1	2
18	how do I compare strings in java	NF	1	1	NF
19	how to split a string in java	1	1	10	9
20	how to create a file and write to a file in java	NF	3	4	8
21	how can I initialise a static map	2	2	1	4
22	iterating through a collection, avoiding concurrent modification exception when removing in loop	5	NF	10	2
23	how can I generate an md5 hash	4	1	1	1
24	get current stack trace in java	1	1	1	1
25	sort arraylist of custom objects by property	2	NF	7	8
26	how to round a number to n decimal places in java	1	1	3	2
27	how can I pad an integers with zeros on the left	8	NF	10	2
28	how to create a generic array in java	3	1	NF	2
29	reading a plain text file in java	4	3	3	1
30	a for loop to iterate over enum in java	NF	1	NF	NF
31	check if at least two out of three booleans are true	NF	NF	NF	NF
32	how do I convert from int to string	10	1	4	10
33	how to convert a char to a string in java	6	1	NF	1
34	how do I check if a file exists in java	NF	1	3	1
35	java string to date conversion	NF	1	1	NF
36	convert inputstream to byte array in java	1	1	4	1
37	how to check if a string is numeric in java	2	1	NF	1
38	how do I copy an object in java	NF	5	7	NF
39	how do I time a method's execution in java	NF	1	NF	4
40	how to read a large text file line by line using java	8	NF	NF	1
41	how to make a new list in java	4	1	1	NF
42	how to append text to an existing file in java	1	NF	NF	NF
43	converting iso 8601-compliant string to date	9	NF	5	2
44	what is the best way to filter a java collection	NF	2	NF	NF
45	removing whitespace from strings in java	NF	1	NF	1
46	how do I split a string with any whitespace chars as delimiters	NF	NF	1	2
47	in java, what is the best way to determine the size of an objects	NF	1	NF	NF
48	how do I invoke a java method when given the method name as a string	NF	NF	NF	NF
49	how do I get a platform dependent new line character	NF	NF	NF	NF
50	how to convert a map to list in java	7	1	NF	4

(NF: Not Found within the top 10 returned results, DCS: DeepCS [18], CM: CodeMatcher, CH: CodeHow [41], UNIF [8].)

Table 6. The Queries$_{99}$ Collected from Husain et al. [25] and the Evaluation Results

No.	Query	DCS	CM	CH	UNIF	No.	Query	DCS	CM	CH	UNIF
1	convert int to string	2	1	6	2	51	how to randomly pick a number	6	5	NF	1
2	priority queue	NF	NF	NF	1	52	normal distribution	NF	1	NF	NF
3	string to date	9	1	1	6	53	nelder mead optimize	NF	NF	NF	NF
4	sort string list	1	1	NF	7	54	hash set for counting distinct elements	7	NF	NF	NF
5	save list to file	2	1	NF	NF	55	how to get database table name	NF	2	NF	NF
6	postgresql connection	NF	1	3	NF	56	deserialize json	1	1	1	1
7	confusion matrix	NF	6	NF	2	57	find int in string	5	1	NF	1
8	set working directory	NF	1	NF	1	58	get current process id	6	1	NF	1
9	group by count	NF	2	2	NF	59	regex case insensitive	NF	2	NF	10
10	binomial distribution	NF	7	NF	1	60	custom http error response	7	1	2	NF
11	aes encryption	5	1	3	7	61	how to determine a string is a valid word	NF	NF	NF	NF
12	linear regression	NF	2	4	1	62	html entities replace	NF	NF	1	1
13	socket recv timeout	NF	1	5	1	63	set file attrib hidden	NF	NF	NF	NF
14	write csv	1	2	6	1	64	sorting multiple arrays based on another arrays sorted order	5	NF	7	NF
15	convert decimal to hex	1	1	NF	1	65	string similarity levenshtein	NF	2	6	1
16	export to excel	NF	1	3	4	66	how to get html of website	NF	5	3	NF
17	scatter plot	NF	1	4	NF	67	buffered file reader read text	1	1	1	NF
18	convert json to csv	NF	NF	4	NF	68	encrypt aes ctr mode	NF	NF	1	NF
19	pretty print json	1	1	NF	1	69	matrix multiply	NF	1	1	1
20	replace in file	1	1	NF	2	70	print model summary	NF	NF	NF	NF
21	k means clustering	NF	3	1	3	71	unique elements	1	1	NF	1
22	connect to sql	1	1	1	1	72	extract data from html content	4	NF	2	1
23	html encode string	1	2	4	1	73	heatmap from 3d coordinates	NF	NF	3	NF
24	finding time elapsed using a timer	NF	1	NF	5	74	get all parents of xml node	NF	3	1	NF
25	parse binary file to custom class	NF	NF	NF	NF	75	how to extract zip file recursively	NF	1	7	4
26	get current ip address	2	1	NF	1	76	underline text in label widget	NF	NF	NF	NF
27	convert int to bool	2	1	NF	8	77	unzipping large files	NF	1	2	NF
28	read text file line by line	5	NF	1	1	78	copying a file to a path	NF	1	1	NF
29	get executable path	1	8	1	1	79	get the description of a http status code	5	1	4	1
30	httpclient post json	2	3	1	NF	80	randomly extract x items from a list	1	NF	NF	NF
31	get inner html	4	3	2	2	81	convert a date string into yyyymmdd	1	1	4	2
32	convert string to number	8	1	1	2	82	convert a utc time to epoch	NF	NF	NF	NF
33	format date	1	1	1	1	83	all permutations of a list	NF	1	NF	1
34	readonly array	NF	NF	NF	NF	84	extract latitude and longitude from given input	5	NF	NF	2
35	filter array	2	1	1	1	85	how to check if a checkbox is checked	NF	2	NF	2
36	map to json	2	3	2	3	86	converting uint8 array to image	1	NF	NF	NF
37	parse json file	NF	1	NF	2	87	memoize to disk - persistent memoization	NF	NF	NF	NF
38	get current observable value	NF	NF	NF	NF	88	parse command line argument	1	1	3	1
39	get name of enumerated value	NF	NF	NF	1	89	how to read the contents of a .gz compressed file?	NF	NF	1	NF
40	encode url	1	1	4	3	90	sending binary data over a serial connection	NF	NF	NF	NF
41	create cookie	1	1	NF	NF	91	extracting data from a text file	6	NF	9	1
42	how to empty array	NF	NF	NF	NF	92	positions of substrings in string	1	1	1	6
43	how to get current date	NF	2	1	NF	93	reading element from html - <td>	NF	1	NF	NF
44	how to make the checkbox checked	NF	1	2	NF	94	deducting the median from each column	NF	NF	1	NF
45	initializing array	2	1	10	4	95	concatenate several file remove header lines	NF	1	NF	NF
46	how to reverse a string	NF	1	1	5	96	parse query string in url	2	NF	9	2
47	read properties file	1	1	NF	1	97	fuzzy match ranking	NF	NF	NF	NF
48	copy to clipboard	1	1	1	3	98	output to html file	NF	NF	2	NF
49	convert html to pdf	NF	1	2	NF	99	how to read .csv file in an efficient way	NF	NF	NF	NF
50	json to xml conversion	NF	1	NF	NF						

(NF: Not Found within the top 10 returned results, DCS: DeepCS [18], CM: CodeMatcher, CH: CodeHow [41], UNIF [8].)

Table 7. The Queries$_{25}$ Collected from Mishne et al. [50] and Keivanloo et al. [28] and the Evaluation Results

No.	Query	DCS	CM	CH	UNIF
1	upload a file	1	3	5	1
2	parse a command-line and get values from it	NF	1	1	NF
3	prepare the executable and argument of a command-line	NF	2	NF	NF
4	create a path element and append it to existing and boot paths	NF	NF	NF	NF
5	run a query and iterate over the results	NF	NF	NF	NF
6	commit and then rollback a transaction	NF	2	1	1
7	get the key of an array element type	1	NF	6	NF
8	get the description and nature IDs of a Java project	NF	NF	NF	NF
9	create a new action	2	1	1	6
10	get the input for the current editor	NF	2	NF	NF
11	retrieve arguments from command line	NF	1	1	6
12	check user selection	6	1	NF	1
13	set up a ScrollingGraphicalViewer	NF	NF	NF	NF
14	create a project	1	2	NF	3
15	successfully login and logout	NF	1	NF	NF
16	click an Element	1	1	1	2
17	commit and rollback a statement	NF	NF	NF	3
18	send a HTTP request via URLConnection	6	1	2	1
19	redirect Runtime exec() output with System	NF	NF	NF	NF
20	get OS Level information such as memory	NF	NF	NF	1
21	SSH Connection	NF	5	NF	3
22	download and save a file from network	NF	1	NF	1
23	generate a string-based MD5 hash value	NF	NF	6	2
24	read the content of a HttpResponse object line by line	NF	NF	NF	NF
25	search via Lucene and manipulate the hits	NF	NF	1	NF

(NF: Not Found within the top 10 returned results, DCS: DeepCS [18], CM: CodeMatcher, CH: CodeHow [41], UNIF [8].)

first parses the abstract syntax tree of each Java code file by leveraging the Javaparser[14] library and extracts all necessary method components as the inputs of baselines, such as method name, comment, Javadoc, APIs in the method body, and so on. Note that we do not test these models on DeepCS' original testing data but our new data, because DeepCS provides no raw data (i.e., no source code) but preprocessed data that can be only used by DeepCS. To mitigate the replication difficulty for our study, we provide a replication package[15] to share our codebase and source code of CodeMatcher and baseline models. The compared baseline models are described as follows:

DeepCS, the DL-based model proposed by Gu et al. [18]: We trained DeepCS by re-running the source code[16] and pre-processed training data[17] provided by the authors. To test DeepCS on our codebase, we first did some natural language processing (e.g., stemming) following the description in Gu et al.'s paper [18], then encoded data using DeepCS' vocabulary and saved the encoded data into the required format by using DeepCS' internal APIs in source code.

[14]https://github.com/javaparser/javaparser.
[15]**Replication Package: https://bitbucket.org/ChaoLiuCQ/codematcher**.
[16]https://github.com/guxd/deep-code-search.
[17]https://pan.baidu.com/s/1U$_$MtFXqq0C-Qh8WUFAWGvg.

CodeHow, the IR-based model developed by Lv et al. [41]: It expands a query with words in related official APIs and matches it with code methods. As Lv et al. [41] provide no source code and data for replication, we re-implemented CodeHow strictly following their paper. Our implementation can be found in our shared replication package. Note that as CodeHow was used for searching C# projects while our targets are Java repositories, we thus used the JDK (Java development kit) as the source of official APIs.

UNIF, the DL-based model proposed by Cambronero et al. [8]: Similar to DeepCS, the UNIF transforms code and query into vectors, and it is trained by pairs of code and natural language description. However, the main differences are that: UNIF represents code by method name and a bag of tokens, which is a subset of DeepCS' input; the UNIF used the pre-trained fastText [4] as its embedding layer. Details can be found in Reference [8]. As the author provided no source code, we re-implemented it by ourselves strictly following the description in the original study, which is also provided in our replication package.

4.4 Evaluation Criteria

To measure the effectiveness of code search models, we need to identify the relevancy of a returned code method to a query. Following Gu et al. [18], the relevancy is manually identified by two independent developers, and the disagreements are resolved by open discussions. During the relevance identification, developers only consider the top-10 returned code methods. Based on the identified relevancy, we measure model performance using four widely used evaluation metrics following Gu et al. [18], including FRank, SuccessRate@k, Precision@k, and **Mean Reciprocal Rank (MRR)**. Note that FRank is defined only for one query, while the other metrics are defined on all queries.

FRank, is the rank of the first correct result in the result list [18]. It measures users' inspection efforts for finding a relevant method when scanning the candidate list from top to bottom. A smaller FRank value implies fewer efforts and better effectiveness of a code search tool for a particular query.

SuccessRate@k, the percentage of queries for which more than one correct result exists in the top-k ranked results [18]. Specifically, $SuccessRate@k = Q^{-1}\sum_{q=1}^{Q}\delta(FRank_q \leq k)$, where Q is the total number of tested queries; $\delta(\cdot)$ is an indicator function that returns 1 if the input is true and 0 otherwise. Higher SuccessRate@k means better code search performance, and users can find desired method by inspecting fewer returned method list.

Precision@k, is the average percentage of relevant results in top-k returned method list for all queries. It is calculated by $Precision@k = Q^{-1}\sum_{q=1}^{Q} r_q/k$, where Q is the total number of queries; r_q is the number of related results for a query q [18]. Precision@k is useful and important because users often check many returned results for learning different code usages [55]. Larger Precision@k indicates that a code search model returns less noisy results.

MRR, the average of the reciprocal ranks for all queries, where the reciprocal rank of a query is the inverse of the rank of the first relevant result ($FRank$) [18]. Thus, the formula is $MRR = Q^{-1}\sum_{q=1}^{Q} FRank_q^{-1}$. Larger MRR value means a higher ranking for the first relevant methods.

5 RESULTS

This section provides the experimental results to the four investigated RQs raised in Section 4.1.

Table 8. Performance Comparison of CodeMatcher and Baseline Models (DeepCS, CodeHow, and UNIF), where the Model Performance Is Measured by SuccessRate@1/5/10 (SR@1/5/10), Precision@1/5/10 (P@1/5/10), and MRR

Queries	Model	SR@1	SR@5	SR@10	P@1	P@5	P@10	MRR
$Queries_{50}$	DeepCS	0.22	0.46	0.64	0.22	0.23	0.22	0.36
	CodeHow	0.30	0.52	0.62	0.30	0.23	0.21	0.41
	UNIF	0.22	0.58	0.68	0.22	0.21	0.17	0.40
	CodeMatcher	0.64	0.76	0.76	0.64	0.58	0.57	0.71
$Queries_{99}$	DeepCS	0.20	0.38	0.45	0.20	0.14	0.11	0.33
	CodeHow	0.22	0.45	0.54	0.22	0.21	0.19	0.36
	UNIF	0.33	0.49	0.56	0.33	0.21	0.15	0.43
	CodeMatcher	0.48	0.65	0.68	0.48	0.42	0.37	0.58
$Query_{25}$	DeepCS	0.16	0.20	0.28	0.16	0.06	0.06	0.26
	CodeHow	0.24	0.32	0.40	0.24	0.15	0.14	0.34
	UNIF	0.24	0.44	0.52	0.24	0.22	0.15	0.38
	CodeMatcher	0.28	0.56	0.56	0.28	0.26	0.19	0.46
$Queries_{all}$	DeepCS	0.20	0.38	0.48	0.20	0.15	0.13	0.33
	CodeHow	0.25	0.45	0.54	0.25	0.20	0.19	0.37
	UNIF	0.29	0.51	0.59	0.29	0.21	0.16	0.41
	CodeMatcher	0.50	0.67	0.68	0.50	0.44	0.40	0.60

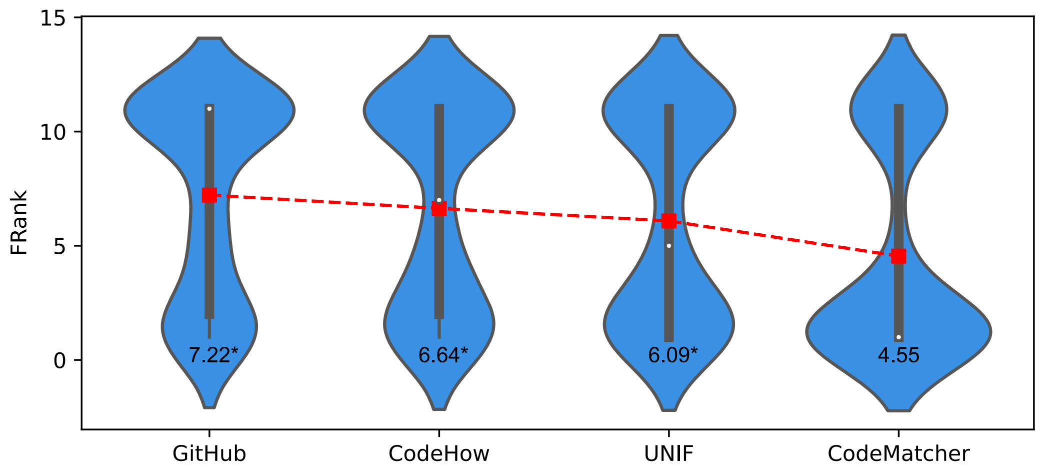

Fig. 2. Violin plots of FRank for CodeMatcher and baseline models (DeepCS, CodeHow, and UNIF), where red squares and white dots indicate the mean and median FRank, respectively; the values denote the mean FRank for each model; "*" indicates the significant difference (p-value < 0.05) between a model and CodeMatcher, which is tested by the Wilcoxon signed-rank test [69] at a 5% significance level.

5.1 Can CodeMatcher Outperform the Baseline Models?

Table 8 compares the experimental results between the proposed model CodeMatcher and three baseline models (i.e., DeepCS, CodeHow, and UNIF) on our large-scale testing data. We can notice that, for the studied 174 queries ($Queries_{all}$), DeepCS obtains an MRR of 0.33, where SuccessRate@1/5/10 = 0.20/0.38/0.48 and Precision@1/5/10 = 0.20/0.15/0.13. Meanwhile, CodeHow achieves an MRR of 0.37 with SuccessRate@1/5/10 = 0.25/0.45/0.54 and

Precision@1/5/10 = 0.25/0.20/0.19. Besides, UNIF gains a better search performance with MRR = 0.41, where SuccessRate@1/5/10 = 0.29/0.51/0.59 and Precision@1/5/10 = 0.29/0.21/0.16.

Table 8 shows that the proposed model CodeMatcher achieves an MRR of 0.60 with SuccessRate@1/5/10 = 0.50/0.67/0.68 and Precision@1/5/10 = 0.50/0.44/0.40. Comparing with the baseline models (DeepCS, Codehow, and UNIF), the MRR improved by 81.8%, 62.2%, and 46.3%, respectively; SuccessRate@1/5/10 increased by 150%/76.3%/41.7%, 100%/48.9%/25.9%, and 72.4%/31.4%/15.3%, respectively; Precision@1/5/10 boosted by 150%/193.3%/207.7%, 100%/120%/110.5%, and 72.4%/109.5%/150%, respectively. Moreover, Figure 2 shows the violin plots of FRank between models, where CodeMatcher obtains better mean FRank (4.55) over the other baselines (>6). CodeMatcher's advantage in FRank is also statistically significant (p-value < 0.05), after we tested the FRank between CodeMatcher and a model with the Wilcoxon signed-rank test [69] at a 5% significance level. The above experimental results imply that the CodeMatcher performs well and clearly outperforms existing solutions.

Result 1: *Our CodeMatcher outperforms the baseline models (DeepCS, CodeHow, and UNIF) substantially, indicating the simplification from DeepCS to CodeMatcher is reasonable and valuable.*

5.2 Is CodeMatcher Faster than the Baseline Models?

Table 9 compares the time efficiency between CodeMatcher and baseline models (DeepCS, CodeHow, and UNIF). All these models ran on a server with an Intel-i7 CPU and an Nvidia Geforce GTX1060 GPU. In the practical usage, developers expect that code search models can quickly respond their search query, namely, short searching time. For the IR-based models (CodeMatcher and CodeHow), they can quickly retrieve a small subset of methods from codebase by leveraging the indexing technique (commonly with <1 s for a query) and sort the candidates with a lightweight ranking strategy. Note that the indexing technique needs to build indexing for all methods in advance to facilitate the code search. Although building indexing may take a long time, it is acceptable as building indexing commonly works offline. Meanwhile, to return the top-n relevant methods for a query, the DL-based models (DeepCS and UNIF) have to compute the cosine similarities between query and all methods among codebase in terms of high-dimension vectors. As DL-based models cannot accelerate the search time like the IR-based models with the indexing technique, they have to leverage the multi-threading technique to boost their performance.

The results show that DeepCS took 58.16 h to train, 24.51 h to preprocess codebase (i.e., parse, encode, and vectorize method name/APIs/tokens), and 376.4 s to search code for each query. In contrast, UNIF only spent 4.1 h for model training, because its network is simple, which contains only embedding and attention layers instead of the complex LSTM layer used in DeepCS. However, UNIF required 455.3 s to complete the searching task for a query, much slower than DeepCS, because UNIF represented code/query into vector with 500 dimensions [8] while DeepCS only needed vectors with 400 dimensions [18]. Although the dimension just increased by 25%, UNIF takes 21% more time for code search, as the codebase is large with more than 16 million code, as described in Table 4. Therefore, the similarity computation for the higher dimensional vectors takes much more time, even if the UNIF has a simple network.

Comparing with DeepCS and UNIF, our IR-based model CodeMatcher ran faster and substantially decreased the code search time from 370+ s to 0.3 s per query. We can notice that CodeMatcher did not need a long-time training as DL-based models, although the training usually happens rarely (or just once). As to the searching time, CodeMatcher only required 23.5 h to preprocess code (23.2 h for code parsing and 0.3 h for code indexing). We found that the DL-based models work slowly mainly because it is time-consuming (>300 s for a query) to load the 23+GB

Table 9. Time Efficiency Comparison between CodeMatcher and Baseline Models in Three Different Phases

Model	Train	Preprocess	Search
DeepCS	58.2 h	24.5 h	376.4 s/query
UNIF	4.1 h	24.2 h	455.3 s/query
CodeHow	-	23.5 h	2.4 s/query
CodeMatcher	-	23.5 h	0.3 s/query

vectorized codebase (with 16M methods) for computing their cosine similarities with the query even using the multi-threading technique. But the IR-based models do not have this issue, because they leveraged the indexing technique to quickly (<1 s for a query) reduce the search space scale (from 16M methods to hundreds of methods or less) so the code search can be done in a very short time. Therefore, further DL-based studies need to consider ways to solve this bottleneck. Additionally, CodeMatcher works eight times faster than the IR-based model CodeHow mainly because the fuzzy search component in CodeMatcher only retrieved a limited number of candidate code from codebase as described in Section 3.2.

The 23.5 hours of code preprocessing seems time-consuming for CodeMatcher. However, there are about 17 million methods, as shown in Table 9, and each method only takes about 0.005 s for code preprocessing on average. The low preprocessing time for the IR-based models means that the model can quickly parse and index the update or added methods in codebase. Thus, CodeMatcher can support the dynamic and rapidly expanding code scale of GitHub, as this model requires no optimization, where changed or new methods can be rapidly parsed and indexed.

Result 2: *Our model CodeMatcher is faster than the DL-based models DeepCS and UNIF, because CodeMatcher requires no model training and it can process a query with more than 1.2K times speedup. Meanwhile, the CodeMatcher works eight times faster than the IR-based model CodeHow.*

5.3 How Do the CodeMatcher Components Contribute to the Code Search Performance?

Generally, the CodeMatcher consists of three components: query understanding, fuzzy search, and reranking, as described in Section 3.2. The query understanding aims to collect metadata for fuzzy search and reranking. Note that the query understanding component leveraged the Stanford Parser to identify the property of query words (e.g., noun and verb). However, the property cannot be precisely identified due to the limitation of the Stanford Parser. We found that the parser failed to work for 124 words in 7 cases, which affects 55.2% of the 174 queries. To investigate how the parser's accuracy affects the model performance. We manually corrected the wrong cases and applied them to the code search (CodeMatcher+SP$^+$). Table 10 shows that CodeMatcher+SP$^+$ obtains an MRR of 0.61, outperforming CodeMatcher by only 2%. Figure 3 also indicates that the mean FRank (4.47 vs. 4.55) between CodeMatcher+SP$^+$ and CodeMatcher is not significantly different (p-value $>$ 0.05) after performing the Wilcoxon signed-rank test [69] at a 5% significance level. We found that the optimization seldom improved the search performance because of two reasons. One is that some failed identifications do not influence the importance level for words as described in Table 2, including "Noun->Verb" and "Verb->Noun." The other reason is that many queries are short with a limited number of words so the other two components can still find the expected code. Thus, under this situation, the accuracy of the used Stanford Parser is acceptable for CodeMatcher.

Table 10. Performance Comparison of CodeMatcher with Different Component Settings (-Rerank, Excluding the Reranking Component; -S$_{body}$, Excluding the Ranking Strategy S_{body} in the Reranking Component; +SP$^+$, Using the Corrected Results Generated by the Stanford Parser), where the Model Performance is Measured by SuccessRate@1/5/10 (SR@1/5/10), Precision@1/5/10 (P@1/5/10), and MRR

Queries	Model	SR@1	SR@5	SR@10	P@1	P@5	P@10	MRR
$Queries_{50}$	CodeMatcher	0.64	0.76	0.76	0.64	0.58	0.57	0.71
	CodeMatcher-Rerank	0.54	0.68	0.68	0.54	0.42	0.40	0.63
	CodeMatcher-S_{body}	0.60	0.72	0.74	0.60	0.53	0.50	0.68
	CodeMatcher+SP^+	0.66	0.76	0.76	0.66	0.59	0.58	0.72
$Queries_{99}$	CodeMatcher	0.48	0.65	0.68	0.48	0.42	0.37	0.58
	CodeMatcher-Rerank	0.33	0.51	0.58	0.33	0.29	0.25	0.45
	CodeMatcher-S_{body}	0.45	0.62	0.67	0.45	0.37	0.33	0.56
	CodeMatcher+SP^+	0.49	0.66	0.69	0.49	0.42	0.38	0.60
$Queries_{25}$	CodeMatcher	0.28	0.56	0.56	0.28	0.26	0.19	0.46
	CodeMatcher-Rerank	0.12	0.16	0.24	0.12	0.11	0.09	0.21
	CodeMatcher-S_{body}	0.32	0.52	0.52	0.32	0.28	0.22	0.46
	CodeMatcher+SP^+	0.32	0.56	0.56	0.32	0.30	0.26	0.48
$Queries_{all}$	CodeMatcher	0.50	0.67	0.68	0.50	0.44	0.40	0.60
	CodeMatcher-Rerank	0.36	0.51	0.56	0.36	0.30	0.27	0.47
	CodeMatcher-S_{body}	0.48	0.63	0.67	0.48	0.40	0.37	0.58
	CodeMatcher+SP^+	0.52	0.67	0.69	0.52	0.45	0.42	0.61

Table 11. Seven Cases that the Stanford Parser in CodeMatcher Generated Wrong Property For Query Words, e.g., "noun->verb" Means a Query Word is Wrongly Identified as a Noun Instead of a Verb

Wrong Case	No.	Wrong Case	No.
Noun->Verb	67	Noun->Adjective	7
Verb->Noun	23	Verb->Adjective	2
Adjective->Verb	18	Conjective->Noun	1
Adjective->Noun	6	**Total**	124

Moreover, to investigate how the key components (fuzzy search and reranking) affect the model performance, we tested the CodeMatcher by excluding the reranking (CodeMatcher-Rerank). Table 10 shows that the MRR of CodeMatcher-Rerank is 0.47, which is reduced by 21.7% comparing with the MRR of CodeMatcher. Besides, Figure 3 shows that the mean FRank value is increased from 4.55 to 6.0, where the difference is significant (p-value < 0.05). These results indicate that the reranking plays an important role in searching code. Besides, we can observe that even though the fuzzy search component outperforms the other baseline models (DeepCS, CodeHow, and UNIF) in terms of MRR (<0.42) by 42.4%, 27.0%, and 14.7%, respectively.

For CodeMatcher, described in Section 3, S_{name} and S_{body} are two matching scores to determine the ranks of searched results, but it is unknown how much they attributed to the performance of CodeMatcher. Thus, we used CodeMatcher for code search but removing the matching score S_{body}. Table 8 shows that the model only using S_{name} (CodeMatcher-S_{body}) obtains MRR = 0.47. Comparing with standard CodeMatcher, MRR is reduced by 21.7%. Besides, the violin plots in Figure 3 shows that the mean FRank (4.55 vs. 4.81) between CodeMatcher and CodeMatcher-S_{body} are not significantly different (p-value > 0.05 for the Wilcoxon signed-rank test [69] at a 5% significance

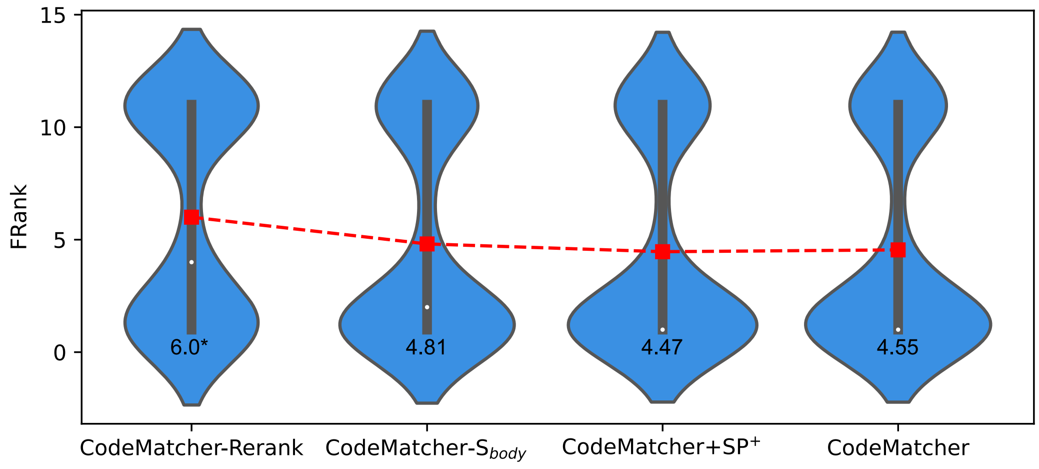

Fig. 3. Violin plots of FRank for CodeMatcher with different settings (-Rerank, excluding the reranking component; -S$_{body}$, excluding the ranking strategy S_{body} in the reranking component; +SP$^+$, using the corrected results generated by the Stanford Parser), where red squares and white dots indicate the mean and median FRank, respectively; the values denote the mean FRank for each model; "*" indicates the significant difference (p-value < 0.05) between a model and CodeMatcher, which is tested by the Wilcoxon signed-rank test [69] at a 5% significance level.

level). These results indicate that the score S_{name} that matches query keywords with method name dominated the performance of CodeMatcher and implies that method name is a significant bridge for the semantic gap between query and code. Furthermore, although the influence of S_{body} that matches query keywords with method body is low, we cannot ignore its contribution. We observed that S_{body} did not work as well as S_{name}, because it cannot fully connect the semantics of query and method body, which are written in natural language and programming language, respectively. Therefore, the S_{body} part requires further improvement.

Result 3: *All the CodeMatcher components are necessarily required. CodeMatcher works well mainly because it can precisely match the query with relevant methods in names by considering the importance of the programming words in the query and the order of query tokens.*

5.4 Can CodeMatcher Outperform Existing Online Code Search Engines?

GitHub[18] and Google[19] search engines are what developers commonly used for code search in the real world. Comparing CodeMatcher with GitHub/Google search is helpful for understanding the usefulness of CodeMatcher. To investigate the advantages of CodeMatcher over the GitHub/Google search, we used the 174 queries in Tables 5–7 as their inputs and performed code search on the whole Java code methods in GitHub. To work on GitHub codebase, the Google search engine performs code search with the following advanced settings: "site:github.com" and "filetype:java."

However, we need to note that GitHub and Google search engines are different from CodeMatcher in three aspects: *(1) Larger-scale codebase.* The codebase of GitHub and Google contains more repositories, because these two online search engines cannot control the search scope as our

[18]https://github.com/search.
[19]https://www.google.com/.

Table 12. Performance Comparison of CodeMatcher and Online Search Engines (Google and GitHub Search), where the Model Performance is Measured by SuccessRate@1/5/10 (SR@1/5/10), Precision@1/5/10 (P@1/5/10), and MRR

Queries	Model	SR@1	SR@5	SR@10	P@1	P@5	P@10	MRR
Queries$_{50}$	CodeMatcher	0.64	0.76	0.76	0.64	0.58	0.57	0.71
	GitHub Search	0.28	0.60	0.64	0.28	0.21	0.17	0.44
	Google Search	0.32	0.80	0.90	0.32	0.34	0.30	0.51
	CodeMatcher+Google	0.72	0.90	0.94	0.72	0.42	0.34	0.80
Queries$_{99}$	CodeMatcher	0.48	0.65	0.68	0.48	0.42	0.37	0.58
	GitHub Search	0.27	0.57	0.64	0.27	0.21	0.17	0.42
	Google Search	0.27	0.56	0.66	0.27	0.29	0.24	0.42
	CodeMatcher+Google	0.49	0.71	0.76	0.49	0.34	0.26	0.61
Queries$_{25}$	CodeMatcher	0.28	0.56	0.56	0.28	0.26	0.19	0.46
	GitHub Search	0.16	0.40	0.48	0.16	0.14	0.14	0.31
	Google Search	0.28	0.52	0.60	0.28	0.24	0.24	0.41
	CodeMatcher+Google	0.32	0.56	0.68	0.32	0.25	0.24	0.45
Queries$_{all}$	CodeMatcher	0.50	0.67	0.68	0.50	0.44	0.40	0.60
	GitHub Search	0.26	0.55	0.61	0.26	0.20	0.17	0.41
	Google Search	0.29	0.62	0.72	0.29	0.30	0.26	0.45
	CodeMatcher+Google	0.53	0.74	0.80	0.53	0.35	0.28	0.64

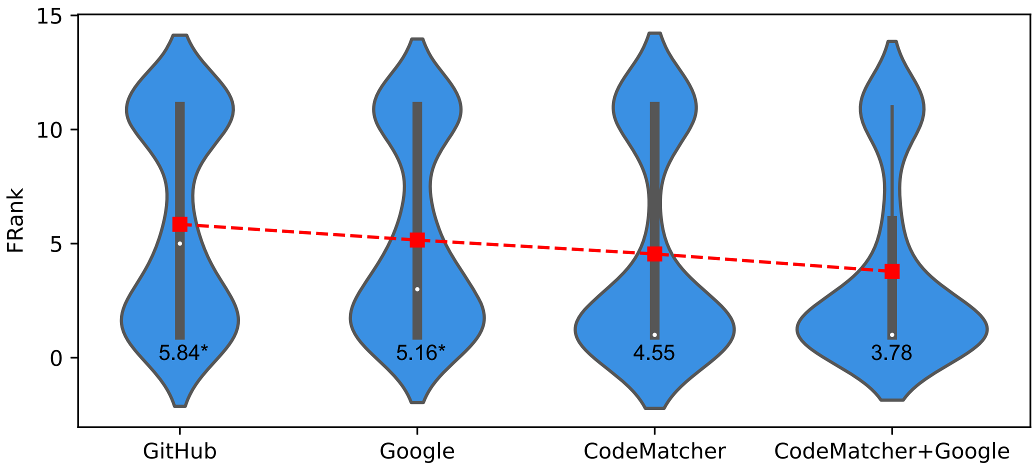

Fig. 4. Violin plots of FRank for CodeMatcher and compared models (GitHub search, Google search, CodeMatcher + Google), where red squares and white dots indicate the mean and median FRank, respectively; the values denote the mean FRank for each model; "*" indicates the significant difference (p-value < 0.05) between a model and CodeMatcher, which is tested by the Wilcoxon signed-rank test [69].

collected codebase for CodeMatcher. *(2) Wider context.* GitHub/Google search matches keywords in a query on all text in code files (e.g., method, comment, and Javadoc), while CodeMatcher only uses texts on methods. *(3) Code snippet vs. method.* GitHub/Google search returns a list of code snippets, but they would not necessarily be a method as what CodeMatcher returns. When using the GitHub and Google search engines for code search, both of them return a list of code snippets with matched keywords, and then we inspect top-10 code snippets to check whether the snippets are relevant to the corresponding queries. We excluded the snippet that was returned just because its comment records the Stack Overflow link (e.g., *"https://stackoverflow.com/questions/157944/create-arraylist-from-array"*) with the same string as the search query (e.g., "create arraylist from array"). This is because the query is also collected from the question title of that link, so this kind of code snippet will overestimate the performance of GitHub/Google search. As CodeMatcher searches code not based on comment or Javadoc, we do not need to filter the Stack Overflow links as GitHub and Google searchers.

Table 12 shows that the GitHub search achieves MRR = 0.41 with SuccessRate@1/5/10 = 0.26/0.55/0.61 and Precision@1/5/10 = 0.26/0.20/0.17. Meanwhile the Google search obtains an MRR of 0.45, where SuccessRate@1/5/10 = 0.29/0.62/0.72 while Precision@1/5/10 = 0.29/0.30/0.26, respectively. We can notice that CodeMatcher outperforms GitHub and Google search engines by 46.3% and 33.3%, respectively, in terms of MRR; by 92.3%/21.8%/11.5% and 72.4%/8.1%/-5.6% in terms of SuccessRate@1/5/10; by 92.3%/120%/135.3% and 72.4%/46.7%/53.8% in terms of Precision@1/5/10. Additionally, as illustrated in Figure 4, CodeMatcher achieves better mean FRank (4.55) over GitHub/Google search (mean FRank > 5.1); the difference is statistically significant (p-value < 0.05 tested by the Wilcoxon signed-rank test [69] at a 5% significance level). These results indicate that CodeMatcher shows advantages in SuccessRate@1/5, Precision@1/5/10, and MRR as compared with the existing two online search engines. These experimental results indicate the practical merit of the CodeMatcher over the GitHub/Google search.

Moreover, we can notice that although Google cannot achieve high precision as CodeMatcher if returning only one method for a query, it can successfully recommend at least one relevant method for more queries. Due to this observation, we investigate the code search performance of combining CodeMatcher and Google together, where the first method returned by Google is replaced by the one recommended by CodeMatcher. The last row in Table 12 shows that the combined model (CodeMatcher+Google) gains the best SuccessRate@1/5/10 (0.53/0.74/0.80), Precision@1 (0.53), and MRR (0.64) compared with other models in the table, even if there are some sacrifices in Precision@5/10 (0.35/0.28). We found that CodeMatcher+Google gains higher SuccessRate@1 than CodeMatcher, because CodeMatcher cannot find any results for some queries that can be compensated by Google's results. Specifically, the CodeMatcher and Google cannot return any correct code for 31.6% and 28.2% of total queries, respectively, when we inspected the top-10 returned results. But the adopted combination strategy (CodeMatcher+Google) can reduce the failure rate to 20%. We can also notice that the combined model (CodeMatcher+Google) improved the mean FRanks of CodeMatcher (4.55) and Google (5.16) to 3.78 as illustrated in Figure 4, although the improvement over CodeMatcher is not statistically significant (p-value > 0.05 tested by the Wilcoxon signed-rank test [69] at a 5% significance level). This case implies that it is beneficial to incorporate the proposed model CodeMatcher into the Google search engine.

Result 4: *CodeMatcher is also advantageous over existing online search engines, GitHub and Google search. It is beneficial to incorporate CodeMatcher into the Google search for practical usage.*

6 DISCUSSION

6.1 Qualitative Analysis

This subsection performs a qualitative analysis of CodeMatcher, DeepCS, CodeHow, and UNIF. To compare these models, we classified all their returned methods into seven categories, namely, whether the query can match the semantics of method name/body. Table 13 illustrates that the definitions and real query-code examples for each category. To be specific, the categories identified whether the method name or body of a searched code can reflect the semantics in query; whether a code's method body is useless, i.e., an abstract method or a getter/setter; whether a method is a duplication of previously inspected one in the top-10 code list. Table 14 lists the classification results for different models.

6.1.1 The Reasons Why DeepCS and UNIF Succeeded and Failed. From Table 14, we observed that the DeepCS/UNIF obtained a 13%/15.9% success (MM and NM), where 9.1%/11.8% of success in MM was due to a correct semantic matching between query and method, as Table 13(1); the 3.9%/4.1% of success in NM implies that DeepCS/UNIF can somewhat capture the semantics in code (i.e., API sequence and tokens) although the method name does not relate to the goal of a query as Table 13(2). However, there are 87%/88.2% of failed results, where 0.1%/0.0% of failures were caused by **returning repeated methods (RM)**. The source code provided by Gu et al. [18] excluded the methods whose cosine similarity differences with related queries are larger than 0.01. But we observed that this judgment could not clear out repeated methods for the DL-based models with some negligible difference, e.g., the modifier difference, as shown in Table 13(7). Meanwhile, 1.4%/0.4% of failures (MN) were caused by unmatched method body, because two methods for different usages may have the same method name, as exemplified in Table 13(3).

Moreover, for the 10.7%/7.3% of failures (MU and NU), we found that DeepCS/UNIF returned some useless methods that can be a getter/setter for a class, or contain abstract APIs with insufficient context to understand, such as the examples in Table 13(4–5). In this way, useless methods do not satisfy the requirement of the method-level code search, since developers need to search and jump to related code. And the manual code jump will increase developers' code inspection time, and it is also uncertain how many jumps they need. Thus, the self-contained source code is advantageous for the method-level code search. In addition, for the most part (71.8%/76.3%) of features (NN), DeepCS/UNIF completely mismatched the code to queries, as illustrated in Table 13(6). We attribute these failures to the insufficient model training, because (1) DeepCS/UNIF was optimized by pairs of method and Javadoc comment, but 500 epochs of training with randomly selected pairs cannot guarantee its sufficiency; (2) DeepCS/UNIF assumed that the first line of Javadoc comment could well describe the goal of related code, but it is uncertain whether the used line is a satisfactory label or a noise; (3) During the model training, the optimization never stops because of the convergence of its loss function values.

Observation 1: *The DL-based models (DeepCS and UNIF) can capture the semantics between queries and code methods. But its unsatisfactory performance is likely derived from the gap between the training data and our codebase.*

6.1.2 CodeMatcher vs. DeepCS/UNIF. For the proposed model CodeMatcher that matches keywords in query with a method, Table 14 shows that 40.2% of code search succeeded due to the well-matched method name and body. However, there is no success from NM, because wrongly combining keywords in the query only leads to unmatched code, i.e., MN (5.3%) and NN (45.3%). This is because CodeMatcher cannot handle complex queries like DeepCS/UNIF via the embedding technique. The 8.4% of failures (MU and NU) on useless methods indicate that boosting the useful methods on a higher rank in terms of the percentage of JDK APIs is not the optimal solution, and directly removing those useless methods may be a better substitution. Moreover, we can observe that as CodeMatcher searched code methods based on how the tokens in methods matched the important query words sequentially, this sequential requirement would substantially exclude many

Table 13. Definitions of Seven Categories on Search Results and their Query-code Examples

1. MM: Matched method name and Matched method body.

Query: create arraylist from array

```
public void createArrayListFromArray(){
    String[] dogs = {“Puppy”, “Julie”, “Tommy”};
    List<String> doglist = Arrays.asList(dogs);
    assertEqual(3, dogsList.size());
}
```

2. NM: Not-matched method name but matched method body.

Query: how to declare an array

```
public void arrayCardinality(ParameterRegistration parameterRegistration){
    Integer[] array = new Integer[]{1, 2, 3};
    Int arrayCardinality = procedures(parameterRegistration).arrayCardinality(array);
    assertEquals(array.length, arrayCardinality);
}
```

3. MN: Matched method name but Not-matched method body.

Query: converting string to int in java

```
static Int convertStatusStringToInt(String statusVal){
    if (statusVal.equalsIgnoreCase(STATUS_REGRESSION)) ||
        statusVal.equalsIgnoreCase(STATUS_FAILED){
        return -1;
    } else if (statusVal.equalsIgnoreCase(STATUS_PASSED)){
        return 1;
    }
    return 0;
}
```

4. MU: Matched method name but Useless method body.

Query: how can I initialize a static map

```
private void initialiseMap(GoogleMap googleMap){
    mMap = googleMap;
}
```

5. NU: Not-matched method name and Useless method body.

Query: converting iso 8601-compliant string to date

```
public static String convertDate2String(Date date){
    return convertDate2String(date);
}
```

6. NN: Not-matched method name and Not-matched method body.

Query: how to read a large text file line by line using java

```
private String textLine(String name, long version, String value){
    return String.format(“name: %s version: %d value: %s,” name, version, value);
}
```

7. RM: Repeated Method.

Query: convert an inputstream to a string

```
public static String convertInputStreamToString(InputStream inputStream){...}
private String convertInputStreamToString(InputStream inputStream){...}
```

Table 14. Classification of 1,740 Code Search Results into 7 Categories for Different Models

Queries	Model	MM	NM	MN	MU	NU	NN	RM
$Queries_{50}$	DeepCS	93 (18.6%)	12 (2.4%)	25 (5.0%)	53 (10.6%)	97 (19.4%)	218 (43.6%)	2 (0.4%)
	CodeHow	34 (6.8%)	73 (14.6%)	2 (0.4%)	1 (0.2%)	4 (0.8%)	331 (66.2%)	55 (11.0%)
	UNIF	60 (12.0%)	23 (4.6%)	0 (0.0%)	7 (1.4%)	18 (3.6%)	392 (78.4%)	0 (0.0%)
	CodeMatcher	285 (57.0%)	0 (0.0%)	32 (6.4%)	27 (5.4%)	28 (5.6%)	125 (25.0%)	3 (0.6%)
$Queries_{99}$	DeepCS	59 (6.0%)	48 (4.8%)	0 (0.0%)	3 (0.3%)	69 (7.0%)	811 (81.9%)	0 (0.0%)
	CodeHow	31 (3.1%)	155 (15.7%)	0 (0.0%)	0 (0.0%)	3 (0.3%)	781 (78.9%)	20 (2.0%)
	UNIF	118 (11.9%)	38 (3.8%)	5 (0.5%)	4 (0.4%)	84 (8.5%)	741 (74.8%)	0 (0.0%)
	CodeMatcher	367 (37.1%)	0 (0.0%)	52 (5.3%)	89 (8.9%)	0 (0.0%)	473 (47.8%)	9 (0.9%)
$Queries_{25}$	DeepCS	6 (2.4%)	8 (3.2%)	0 (0.0%)	1 (0.4%)	15 (6.0%)	220 (8.0%)	0 (0.0%)
	CodeHow	6 (2.4%)	29 (11.6%)	0 (0.0%)	0 (0.0%)	0 (0.0%)	213 (85.2%)	2 (0.8%)
	UNIF	28 (11.2%)	10 (4.0%)	2 (0.8%)	0 (0.0%)	15 (6.0%)	195 (78.0%)	0 (0.0%)
	CodeMatcher	47 (18.8%)	1 (0.4%)	8 (3.2%)	3 (1.2%)	0 (0.0%)	191 (76.4%)	0 (0.0%)
$Queries_{all}$	DeepCS	158 (9.1%)	68 (3.9%)	25 (1.4%)	57 (3.3%)	181 (10.4%)	1,249 (71.8%)	2 (0.1%)
	CodeHow	71 (4.1%)	257 (14.8%)	2 (0.1%)	1 (0.1%)	7 (0.4%)	1,325 (76.1%)	77 (4.4%)
	UNIF	206 (11.8%)	71 (4.1%)	7 (0.4%)	11 (0.6%)	117 (6.7%)	1,328 (76.3%)	0 (0.0%)
	CodeMatcher	699 (40.2%)	1 (0.1%)	92 (5.3%)	119 (6.8%)	28 (1.6%)	789 (45.3%)	12 (0.7%)

duplicated methods for a query. After filtering out the redundant methods by simply comparing their MD5 hash values, the CodeMatcher can only return a limited number of repeated methods (RM = 12). Comparing with DeepCS/UNIF, CodeMatcher returned more repeated methods. Thus, simply filtering redundant methods by their MD5 hash values, as described in Section 3, is not enough. A better choice would be comparing the API usages, data structure, and working flow in the method body. Comparing with DeepCS/UNIF, the main advantage of CodeMatcher is the correct keywords matching between query and code, i.e., a high percentage of MM. However, CodeMatcher can hardly handle partial matching only on the code body (i.e., NM = 1). But this is what DL-based models are good at (NM = 68 for DeepCS and NM = 71 for UNIF), because they can capture high-level intent between query and method by joint embedding, although the NM are much smaller than MM for DeepCS/UNIF (MM = 158 for DeepCS while MM = 206 for UNIF). Therefore, the advantages of DeepCS/UNIF are good to compensate for CodeMatcher's disadvantages.

Observation 2: *CodeMatcher cannot address complex queries like DL-based models (DeepCS and UNIF). Meanwhile, CodeMatcher can accurately find code methods for simple queries and avoid the out-of-vocabulary issue in DeepCS and UNIF.*

6.1.3 CodeMatcher vs. CodeHow. By analyzing the classification of these search results in Table 14, we can observe that CodeHow is good at matching a query with the method body (14.8% for NM), because it extends query with related official APIs so method body can be well filtered in terms of internal APIs. However, the main reason for the failures is the unmatched keywords (76.1% for NN), since CodeHow ignores the importance of programming words and their sequence. The other main reason is that it does not exclude repeated methods (4.4% for RM). However, we can observe that CodeHow can compensate for the disadvantages of CodeMatcher, i.e., the difficulty in matching a query with the method body.

Observation 3: *The IR-based models CodeMatcher and CodeHow compensate with each other, as they are good at matching a query with method name and body, respectively.*

6.2 Conciseness and Completeness

Keivanloo et al. [28] indicated that, for a searched code, although the correctness (i.e., whether it is relevant to the search query) is important, developers also care about two other features. One feature is the conciseness, the ratio of irrelevant lines to the total lines. Lower conciseness indicates that a code contains no irrelevant lines. The other feature is the completeness, the number of addressed tasks divided by the total number of tasks, where the task includes the intent of the search query and other missed statements (well-typed, variable initialization, control flow, and exception handling) [28]. Besides, Keivanloo et al. [28] indicated that the code readability (whether the variable names are well-chosen) is also important. However, the readability is not easy to measure and the conciseness is the key component of readability (namely, the searched code should use "as little code as possible" and show "the most basic version of the problem") [7]. This is also the reason why Keivanloo et al. [28] only measured the conciseness and completeness of code.

Figures 5–6 show these two features of all the code searched by the CodeMatcher and baseline models (DeepCS, CodeHow, and UNIF). Figure 5 shows that the average conciseness of DeepCS, CodeHow, and UNIF are 0.88, 0.83, and 0.85, respectively. The proposed model CodeMatcher achieves a value of 0.61, outperforming the baselines by 30.7%, 26.5%, and 28.2%, respectively. After performing the Wilcoxon signed-rank test [69] between the results of CodeMatcher and a baseline model, the statistical result indicates that the advantages of CodeMatcher over baselines are significant (p-value < 0.05). Moreover, Figure 6 shows that CodeMatcher also outperforms baseline models significantly in terms of the completeness with an average value of 0.33.

To investigate why CodeMatcher shows substantially better conciseness and completeness over baselines, we excluded the searched results irrelevant to the 174 queries for each model. As the left searched results are different for CodeMatcher and baselines, we applied the two-sample Kolmogorov-Smirnov test [43] at a 5% significance level to estimate the statistical difference. Figures 7–8 show that the CodeMatcher and baselines have no significant difference on conciseness and completeness for the relevant code. These results indicate that the number of returned code relevant to the query strongly affects the total conciseness and completeness. However, these results also implied that all the studied code search models do not consider the conciseness and completeness in the model. Therefore, we suggest that further studies take efforts to improve the quality of the searched code.

6.3 Why DeepCS and UNIF Did Not Work Well on Our New Dataset

One may notice that the performance of DeepCS and UNIF are lower than the ones reported by Gu et al. [18] and Cambronero et al. [8], respectively. To verify the validity of our re-ran DeepCS and re-implemented UNIF, we tested our trained DeepCS and UNIF not on our new testing data but the original testing data shared by Gu et al. [18]; we refer to the results of our experiment as DeepCS$_{old}$ and UNIF$_{old}$. Table 15 shows that the performance of DeepCS$_{old}$ is close to the one reported by Gu et al. [18] in terms of MRR (0.59 vs. 0.6); the MRR of UNIF$_{old}$ is also close to the one reported by Cambronero et al. [8] (0.54 vs. 0.58). Thus, these results confirm the validity of the re-ran DeepCS model and our re-implemented UNIF model. Moreover, not like Gu et al.'s testing data, our testing data shares no overlap with their training data. Therefore, the reduced performance on our testing data implies that the overlap affects the generalizability of DeepCS and UNIF.

Ideally, the performance of DeepCS and UNIF can be improved by tuning with more training data. To reach the goal, we built the models DeepCS$_{tune}$ and UNIF$_{tune}$ tuned with more training data extracted from our codebase. Totally, we collected 1,283,445 commented code methods as training data instances following the training data extraction process described in Reference [18].

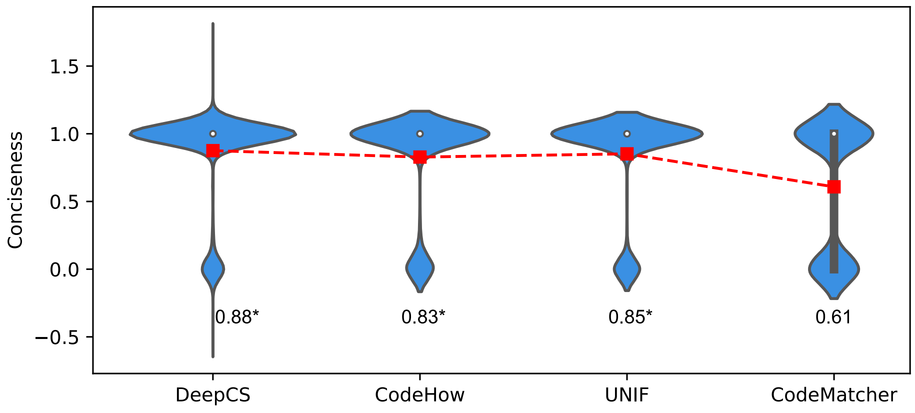

Fig. 5. The conciseness of baseline models (DeepCS, CodeHow, UNIF, and CodeMatcher) for all the searched results; "*" indicates the significant difference (p-value < 0.05) between a model and CodeMatcher, which is tested by the Wilcoxon signed-rank test [69].

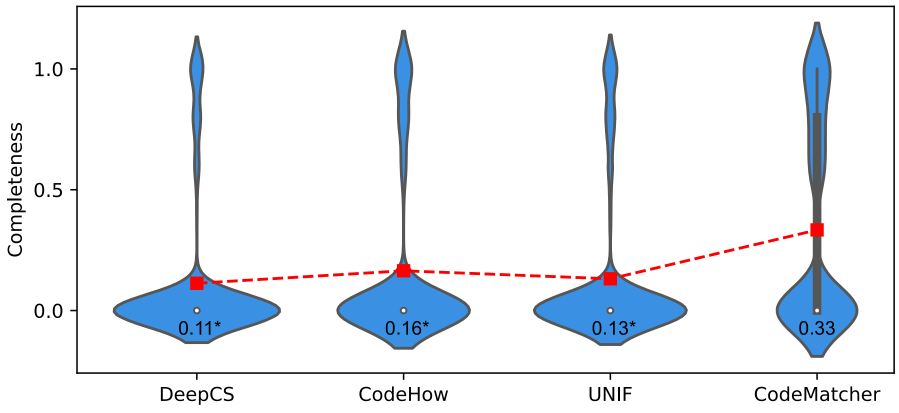

Fig. 6. The completeness of baseline models (DeepCS, CodeHow, UNIF, and CodeMatcher) for all the searched results; "*" indicates the significant difference (p-value < 0.05) between a model and CodeMatcher, which is tested by the Wilcoxon signed-rank test [69].

In the model optimization step, we tuned the pre-trained DeepCS and UNIF with the new training data in default settings, i.e., 500 epochs. Table 15 shows that DeepCS$_{tune}$ achieves a better performance with MRR 0.38 (an improvement of 15.2% over DeepCS), while the MRR of UNIF$_{tune}$ is 0.46 (an improvement of 12.2% over UNIF). The experimental results imply that DeepCS and UNIF did not work due to the gap between Gu et al.'s training and our testing data. Tuning DeepCS and UNIF with data generated from the testing data can mitigate the generalizability issue. However, their tuned performance (MRR = 0.38 for DeepCS and MRR = 0.46 for UNIF) are still far from the result of CodeMatcher (MRR = 0.60).

We observed that this case is likely attributed to the limited training data and a fixed size of vocabulary. As shown in Table 4, our new testing data contains about 16 million methods in total but only 21.91% of them have Javadoc comments. Moreover, only 7.73% of the total methods can be used to tune DeepCS/UNIF due to the limited vocabulary size. Therefore, DeepCS/UNIF would have difficulty in comprehending the semantics of methods and their relationship to

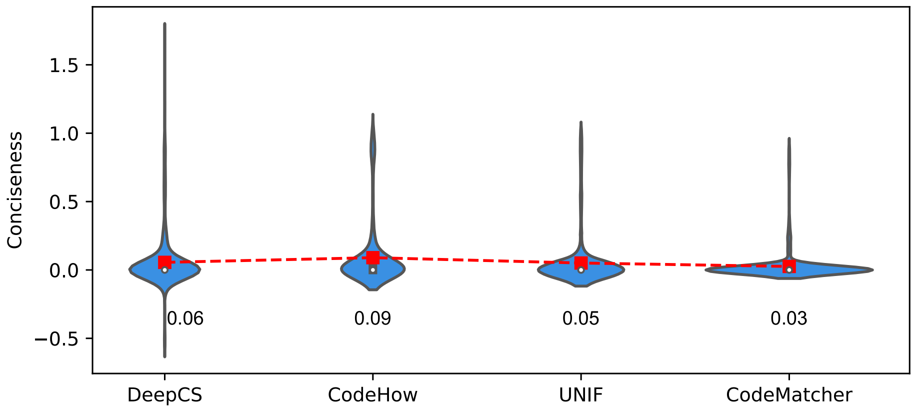

Fig. 7. The conciseness of baseline models (DeepCS, CodeHow, UNIF, and CodeMatcher) for all the correctly searched results; no baseline showed significant difference (p-value > 0.05) with CodeMatcher, which is tested by the two-sample Kolmogorov-Smirnov test [43] at a 5% significance level.

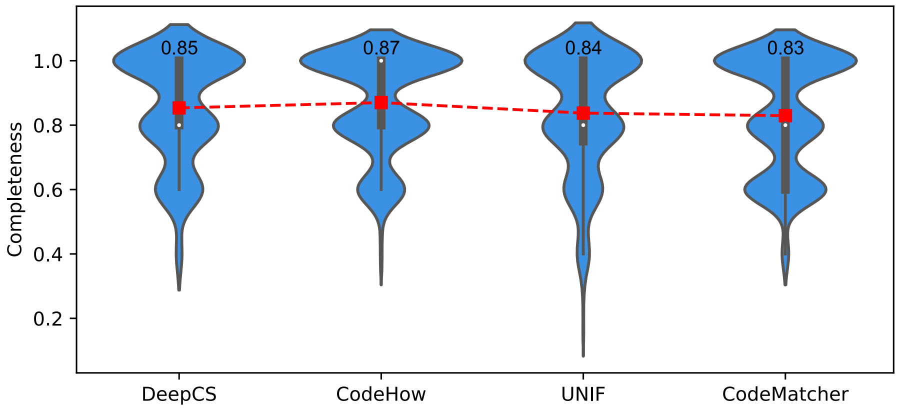

Fig. 8. The completeness of baseline models (DeepCS, CodeHow, UNIF, and CodeMatcher) for all the correctly searched results; no baseline showed significant difference (p-value > 0.05) with CodeMatcher, which is tested by the two-sample Kolmogorov-Smirnov test [43] at a 5% significance level.

related Javadoc comments. To further analyze the impacts of the vocabulary, we investigate how it covers the words in the new testing data. Table 16 shows that more than 95% of words in method components (API, name, and tokens) cannot be covered. In terms of the word frequency, there are 42.95%, 6.22%, and 12.75% of new words that appeared in method API, name, and token, respectively. Due to the out-of-vocabulary issue, more than half (52.91%) of methods in testing data contain new words. Therefore, the trained DeepCS/UNIF would have difficulty in understanding the semantics of methods in the new testing data.

6.4 Why Is the Word Sequence Important?

To capture the sequential relationship between important words in query, the proposed model CodeMatcher considered it in the fuzzy search component and reranking strategies. However, one major question is whether the sequence frequently occurred. To investigate this research question, we analyzed the studied 174 queries. We found that for 79 queries, the order of words that appear

Table 15. Performance Comparison of DeepCS in Different Settings

Query	Model	SR@1	SR@5	SR@10	P@1	P@5	P@10	MRR
$Queries_{50}$	$DeepCS_{old}$	0.44	0.72	0.82	0.44	0.42	0.41	0.59
	$UNIF_{old}$	0.38	0.72	0.80	0.38	0.39	0.33	0.54
$Queries_{50}$	$DeepCS_{tune}$	0.30	0.62	0.72	0.30	0.31	0.30	0.44
	$UNIF_{tune}$	0.26	0.56	0.64	0.26	0.26	0.19	0.41
$Queries_{99}$	$DeepCS_{tune}$	0.24	0.42	0.47	0.24	0.20	0.15	0.36
	$UNIF_{tune}$	0.33	0.64	0.71	0.33	0.32	0.26	0.49
$Queries_{25}$	$DeepCS_{tune}$	0.16	0.36	0.40	0.16	0.18	0.10	0.30
	$UNIF_{tune}$	0.36	0.52	0.52	0.36	0.22	0.17	0.48
$Queries_{all}$	$DeepCS_{tune}$	0.24	0.47	0.53	0.25	0.23	0.19	0.38
	$UNIF_{tune}$	0.32	0.60	0.66	0.32	0.29	0.23	0.46

(old, testing model on DeepCS' original testing data [18]; tune, tuning model with data collected from our testing data and testing model on our testing data), where the model performance is measured by SuccessRate@1/5/10 (SR@1/5/10), Precision@1/5/10 (P@1/5/10), and MRR.

Table 16. Statistics of Words in Our New Testing Data Covered by DeepCS' and UNIF's Vocabulary, where DeepCS Represents Code by API, Name, and Token; Meanwhile, UNIF Represents Code by Name and Token

Item	Count	New	Percentage
Unique Words in API	2,406,547	2,397,969	99.64%
Unique Words in Name	270,084	261,113	96.68%
Unique Words in Token	1,610,147	1,602,435	99.52%
Words in API	64,754,427	27,811,106	42.95%
Words in Name	44,194,936	2,749,420	6.22%
Words in Token	177,572,258	22,633,682	12.75%
Method	16,611,025	8,788,967	52.91%

Table 17. Classification of Cases with Sequentially Important Query Words

No.	Case	Description	Count	Percentage
1	The order of multiple tasks	do A and B	9	11.4%
2	Data conversion	do (from) A to B	38	48.1%
3	Conditional job	do A with/via/in/over/based/by/when/if B	17	21.5%
4	The exchanged core word	do A B	15	19.0%
-	**Total**	-	79	100.0%

in the query plays an important role–if the words in the query are reordered, then the meaning will change. To analyze the root causes, we classified these 79 queries into four cases: *(1) the order of multiple tasks,* the query "read from file A and write to file B" is different from the one "write from file A and read to file," because exchanging the phrases "read from" and "write to" will change the intent of two tasks; *(2) data conversion,* "convert int to string" and "convert string to int" work differently; *(3) conditional job,* "sort map by values" contradicts with the semantics "sort values by map"; *(4) the exchanged core word,* "read property file" and "read file property" would be implemented in two ways, because "property file" and "file property" are two completely different objects. The statistics in Table 17 show that among these four cases, the data conversion is the most frequent one.

7 IMPLICATION

7.1 Pros and Cons of DL-based Model and IR-Based Model

The DL-based model has three major advantages over the IR-based model. One is language processing ability. By leveraging the embedding technique, it can better address complex queries and tolerate errors to some extent according to Section 6. The second is the bilingual learning ability. With the joint embedding framework between a query written in natural language and a code implemented in programming language, their mapping relationship can be well learned, as described in Section 5.1. At last, the DL-based model may require less upfront cost than IR-based model, because the former requires much less domain knowledge and feature engineering like the IR-based model. However, the DL-based models (DeepCS and UNIF) have a limitation on a new and dynamic codebase, because the studied models suffer from overfitting and out-of-vocabulary issues, as discussed in Section 6.3. But these problems do not occur for the IR-based models (CodeHow and CodeMatcher). Meanwhile, running the IR-based model is substantially more efficient over the DL-based model, because the IR-based model requires no model training and can speed up the code search process using a framework like Elasticsearch, as shown in Section 5.2. To sum up, the two kinds of studied code search models complement each other, therefore it is suggested to balance their pros and cons and make a fusion in the future.

Suggestion 1: *Combine the advantages of IR-based and DL-based models.*

7.2 The Importance of Method Name

Section 5.3 indicates that CodeMatcher works well mainly because it can precisely match the semantics between query and the method name, where CodeMatcher assigns higher importance on programming words (e.g., Inputstream or String) and considers their sequence in the query. Besides, Section 6 also shows that if a method precisely matched the semantics in query, then such method is likely to contain expected implementation in the method body. This is because the method name is very similar to the query: *(1) writing in natural language.* There is less semantic gap between query and method name; *(2) short in text.* They usually use the same keywords in order; *(3) specific to code implementation.* Their semantic relationship to code implementation is usually the same and straightforward. Therefore, a code search engine should assign higher weights on method names, no matter for the DL-based or IR-based model.

Moreover, although CodeMatcher is capable of handling synonyms in a query by using the WordNet as described in Section 3, it has three limitations: *(1) abbreviation.* It cannot match the word "initialize" in a query to the method named with "ini"; *(2) Acronym.* The method named with "RMSD" should be missed for a query with the keyword "root mean square deviation"; *(3) low quality of method naming.* The method name is not a correct abstraction on its code implementation. Meanwhile, other code search models (e.g., DeepCS and CodeHow) also do not consider these situations. One solution to these limitations is to increase the scale of the codebase, because the increased search space may include the methods whose names are similar to search queries (e.g., using abbreviations or acronyms). However, to solve these challenges, maybe the best way is to require developers strictly following a method naming standard in the beginning. For example, a developer follows the Google Java style guide[20] and writes method name in verb (phases) with commonly used words, instead of self-defined synonym, acronym, and abbreviation. As to the

[20]https://google.github.io/styleguide/javaguide.html#s5-naming.

existing large-scale source code in GitHub, maybe a better solution is to format their method names in a standard and unified way.

Suggestion 2: *Method name has a significant role in code search; improving the quality of developers' method names helps code search.*

8 THREATS TO VALIDATION AND MODEL LIMITATION

There are some threats affecting the validity of our experimental results and conclusions.

Manual Evaluation. The relevancy of returned code methods to the studied queries was manually identified, which could suffer from subjectivity bias. To mitigate this threat, the manual analysis was performed independently by two developers from Baidu Inc. For the first query set ($Queries_{50}$), they reach a substantial agreement in terms of the value (0.62) of Cohen's Kappa static [67]; and if developers disagreed on a relevancy identification, then they performed an open discussion to resolve it. For the second and third query sets ($Queries_{99}$ and $Queries_{25}$), the agreement was improved to 0.72 in terms of the Cohen's Kappa static due to developers' increased evaluation experiences. In the future, we will mitigate this threat by inviting more developers. Moreover, in the relevancy identification, we only consider the top-10 returned code results following Gu et al. [18]. However, in the real-world code search, this setting is reasonable, because developers would like to inspect the top-10 results and ignore the remaining due to the impacts of developers' time and patience.

Limited Queries and Java Codebase. Following Gu et al. [18], we evaluated the model with popular questions from Stack Overflow, which may not be representative of all possible queries for code search engines. To mitigate this threat, the selected top-50 queries are the most frequently asked questions collected by Gu et al. [18] in a systematic procedure, as referred to in Section 4. We also extended 99 more queries that provided in the CodeSearchNet challenge [25] and 25 more queries from two related studies [28, 50]. Besides, many studied queries are highly related to popular APIs, so the CodeMatcher may not work for queries with less popular APIs. In this case, the performance of CodeMatcher could be overestimated. In the future, we will extend the scale, scope, and variety of the code search queries. We also plan to investigate how to automatically evaluate model performance on a large-scale codebase. Furthermore, we performed the experiments with large-scale open-source Java repositories. But we have not evaluated repositories in other programming languages, though the idea of extending CodeMatcher to any language is easy and applicable. Moreover, we collected about 41K GitHub projects with high-quality code (i.e., more than five stars) as the codebase. But such search space is likely to overestimate a model, because such projects are going to have more accurate Javadoc and generally cleaner, easier to understand code that is more likely to be commented. Although we extended Gu et al.'s [18] codebase (around 1K projects with at least 20 stars) on a larger scale, this situation cannot be ignored. We plan to extend our codebase more in the near future.

Baseline Reproduction. To estimate DeepCS on our testing data, we preprocessed the testing data according to Gu et al. [18], although the source code and training data have no difference. Meanwhile, we re-implemented the baseline CodeHow and UNIF strictly following References [41] and [8], respectively, because their authors provide no source code. Our baseline reproductions may threaten the validity of our model. To mitigate this threat, we double-checked our code and also present all the replication packages in public, as described in Section 4.3.

Model Limitation. Like all the existing code search models, the proposed solution CodeMatcher may not work if a search query shares no irrelevant words with the methods in codebase. To address this limitation, CodeMatcher replaced the words that do not appeared in a codebase by their synonyms extracted from the codebase. This limitation could be further mitigated by combining CodeMatcher with a DL-based model, as described in Section 7.1. We intend to investigate this research in the near future. Section 5.4 indicated the Stanford Parser used in CodeMatcher is not accurate. Although the corrected results only improve the model performance slightly for our studies, we cannot ignore this limitation in the further studies.

9 RELATED WORK

Categories of Code Search. In the software development, developers may directly search existing applications to work on [3], and many application search engines have been built [17, 37, 38, 44, 60, 74]. However, application-level search is not frequently used during the development, and more than 90% of developers' search efforts are used for searching code snippets (e.g., code method) [3]. For this reason, method-level code search has been studied for decades [40, 47, 48, 58], and this study follows this type of code search.

The research objective of method-level code search is to investigate the mechanics of developers' searching behaviors [3, 28, 35, 56, 61, 70] and build a model to fill the semantic gap between natural language (i.e., search query) and programming language (i.e., method source code) [2, 18, 36, 39, 41]. Better code search techniques can boost the rapid software development [46] and promote other search-based researches, such as program synthesis [14, 19], code completion [51, 57, 59], program repair [27, 31], and mining software knowledge [13].

For the setting of method-level code search, this study works on the "query-codebase" search following previous studies [18, 41]. The query in natural language is commonly used as developers' search input, although method declarations and test cases can complement and clarify developers' specification [33, 34, 58]. But we cannot assume that developers would always provide this information. Moreover, the codebase like GitHub is usually used for code retrieval. Because the stored code are large-scale and ready-to-use [2, 18, 41], although there are some useful code provided in Stack Overflow [9] or software development tutorials [54].

Evolution of Method-level Code Search. At the beginning of this study, researchers just regarded code as plain texts and simply applied the capabilities of web search engines into code search [30]. Google Code Search [29], Koders [3], and Krugle [30] were few promising systems [2]. Later, many researchers attributed the challenge of code search to the understanding of query in natural language. To generate correct keywords for a search engine, many existing works focused on query expansion and reformulation [20, 21, 39, 45, 53]. For example, Hill et al. [21] rephrased queries according to the context and semantic role of query words within the method signature. Haiduc et al. [20] proposed Refoqus, which reformulated query by choice of reformulation strategies, and a machine learning model recommends which strategy to use based on query properties. Lu et al. [39] extended a query with synonyms generated from WordNet [32]. McMillan et al. [45] proposed Portfolio, which returns a chain of functions through keyword matching and speeds up search by PageRank [52]. All query processing models can outperform early search engines, such as the substantial improvements of Portfolio over Google Code Search and Koders [45].

However, source code is more than just a plain text, and it contains abundant programming knowledge. Thus, matching code text with keywords is far from enough [2]. To push the code search study forward, Bajracharya et al. [2] proposed Sourcerer, an IR-based code search engine that combines the textual content of a program with structural information. Moreover, researchers observed that the API usage is a key to understand code and its specification. Following this intuition, Li et al. [36] proposed RACS, a search framework for JavaScript that considers API relationships, including their sequencing, condition, callback relationships, and so on. Meanwhile, Lv et al. [41] proposed CodeHow, a code search engine that confers related APIs from a query and matches them with code by an extended Boolean model. Its validity and usefulness were validated by Microsoft developers. Similarly, Feng et al. [72] proposed a model to expand query with semantically related API class names and search the best-matched source code.

Recently, Gu et al. [18] proposed a deep-learning-based model DeepCS, which jointly embeds method in programming language and query in natural languages, and search methods by comparing the similarity between query and candidate methods. Their experiments show that DeepCS can significantly outperform two representative models, Sourcerer [2] and CodeHow [41]. They attributed these improvements to the successful application of a deep learning model, which considers both code property (e.g., API sequence, programming tokens) and text property (e.g., token synonymous, rephrasing, sequencing) in a unified model. Recently, researchers also provided some improved DL-based models. Cambronero et al. [8] developed the model UNIF with much simpler network than DeepCS. Generally, UNIF embeds code and query with embedding layers pre-trained by the fastText [4], and the embedded code and query were attached with attention layer and average layer, respectively. Wan et al. [68] provided a model called **MMAN (Multi-Model Attention Network)** with complex network. Different from DeepCS, MMAN embeds code by a fusion of three neural networks: one LSTM for the sequential tokens of code, a Tree-LSTM for the AST of code, and a **GGNN (Gated Graph Neural Network)** for the control flow graph of code. Feng et al. [12] fine-tuned the heavy-weight DL model BERT [11] on the code search task with pairs of natural language text and code. As described in Section 3, our proposed CodeMatcher can be regarded as a simplified model of DeepCS that also considers code and text property in search but implemented in the way of keyword matching. Experimental results showed that CodeMatcher substantially outperforms DeepCS and UNIF in searching accuracy and time efficiency. Also, CodeMatcher shows better performance over the IR-based model CodeHow.

10 CONCLUSION

The challenge for code search is how to fill the semantic gap between a query written in natural language and a code implemented in programming language. Recently, Gu et al. [18] proposed a DL-based model DeepCS that leverages the DL technique to jointly embed query and method code into a shared high-dimensional vector space, where methods related to a query are retrieved by their vector similarities. However, the working process of DeepCS is complicated and time-consuming. In this article, we proposed a simple and faster model named CodeMatcher, which leverages the IR technique to simplify DeepCS but inherits the advantageous features in DeepCS.

To verify the model validity, we collected a large-scale codebase from GitHub to compare the performance of CodeMatcher and baseline models (DeepCS, CodeHow, and UNIF). Experimental results show that CodeMatcher (MRR = 0.60) substantially outperforms DeepCS, CodeHow, and UNIF by 82%, 62%, and 46%, respectively, in terms of MRR. On the time-efficiency, CodeMatcher is over 1.2K and 1.5K times faster than the DL-based models DeepCS and UNIF. Besides, it works eight times faster than the IR-based model CodeHow. The above experimental results indicate that the CodeMatcher simplified from DeepCS is reasonable and valuable. Besides, we also found that CodeMatcher works well because it can correctly map the semantics of queries to code methods.

Moreover, we also compare CodeMatcher with two existing online code search engines, GitHub and Google search. Experimental results showed that CodeMatcher outperforms these two search engines by 46% and 33%, respectively, in terms of MRR. Comparing with Google search, the major advantage of CodeMatcher is to correctly recommend code method for query in the first place. Thus, the experimental result shows when we incorporated CodeMatcher to Google search, the

MRR can be further improved to 0.64. These results imply the merit of CodeMatcher for practical usage.

Finally, we conducted an in-depth qualitative analysis on results of DL-based and IR-based models and provided some suggestions for code search: It is necessary to combine the advantages of the IR-based and DL-based models together in the future; method name plays a significant role in code search, because it is written in natural language as queries, and improving the quality of method names will help code search.

ACKNOWLEDGMENTS

Any opinions, findings and conclusions or recommendations expressed in this material are those of the author(s) and do not reflect the views of National Research Foundation, Singapore, and Huawei, China. This research is a non-Huawei service achievement.

REFERENCES

[1] Ken Arnold, James Gosling, David Holmes, and David Holmes. 2000. *The Java Programming Language.* Vol. 2. Addison-wesley Reading.

[2] 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 *Proceedings of the 21st ACM SIGPLAN Symposium on Object-oriented Programming Systems, Languages, and Applications.* ACM, 681–682.

[3] Sushil Krishna Bajracharya and Cristina Videira Lopes. 2012. Analyzing and mining a code search engine usage log. *Empir. Softw. Eng.* 17, 4–5 (2012), 424–466.

[4] Piotr Bojanowski, Edouard Grave, Armand Joulin, and Tomas Mikolov. 2017. Enriching word vectors with subword information. *Trans. Assoc. Comput. Ling.* 5 (2017), 135–146.

[5] Joel Brandt, Mira Dontcheva, Marcos Weskamp, and Scott R. Klemmer. 2010. Example-centric programming: Integrating web search into the development environment. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems.* ACM, 513–522.

[6] Joel Brandt, Philip J. Guo, Joel Lewenstein, Mira Dontcheva, and Scott R. Klemmer. 2009. Two studies of opportunistic programming: Interleaving web foraging, learning, and writing code. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems.* ACM, 1589–1598.

[7] Raymond P. L. Buse and Westley Weimer. 2012. Synthesizing API usage examples. In *Proceedings of the 34th International Conference on Software Engineering (ICSE).* IEEE, 782–792.

[8] Jose Cambronero, Hongyu Li, Seohyun Kim, Koushik Sen, and Satish Chandra. 2019. When deep learning met code search. In *Proceedings of the 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering.* 964–974.

[9] Brock Angus Campbell and Christoph Treude. 2017. NLP2Code: Code snippet content assist via natural language tasks. In *Proceedings of the IEEE International Conference on Software Maintenance and Evolution (ICSME).* IEEE, 628–632.

[10] Danqi Chen and Christopher Manning. 2014. A fast and accurate dependency parser using neural networks. In *Proceedings of the Conference on Empirical Methods in Natural Language Processing (EMNLP).* 740–750.

[11] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. 2018. BERT: Pre-training of deep bidirectional transformers for language understanding. *arXiv preprint arXiv:1810.04805* (2018).

[12] Zhangyin Feng, Daya Guo, Duyu Tang, Nan Duan, Xiaocheng Feng, Ming Gong, Linjun Shou, Bing Qin, Ting Liu, Daxin Jiang et al. 2020. CodeBERT: A pre-trained model for programming and natural languages. *arXiv preprint arXiv:2002.08155* (2020).

[13] Mark Gabel and Zhendong Su. 2010. A study of the uniqueness of source code. In *Proceedings of the 18th ACM SIGSOFT International Symposium on Foundations of Software Engineering.* ACM, 147–156.

[14] Joel Galenson, Philip Reames, Rastislav Bodik, Björn Hartmann, and Koushik Sen. 2014. Codehint: Dynamic and interactive synthesis of code snippets. In *Proceedings of the 36th International Conference on Software Engineering.* ACM, 653–663.

[15] Mohammad Gharehyazie, Baishakhi Ray, and Vladimir Filkov. 2017. Some from here, some from there: Cross-project code reuse in GitHub. In *Proceedings of the IEEE/ACM 14th International Conference on Mining Software Repositories (MSR).* IEEE, 291–301.

[16] Clinton Gormley and Zachary Tong. 2015. *Elasticsearch: The Definitive Guide: A Distributed Real-time Search and Analytics Engine.* O'Reilly Media, Inc.

[17] Mark Grechanik, Chen Fu, Qing Xie, Collin McMillan, Denys Poshyvanyk, and Chad Cumby. 2010. A search engine for finding highly relevant applications. In *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering*. ACM, 475–484.

[18] Xiaodong Gu, Hongyu Zhang, and Sunghun Kim. 2018. Deep code search. In *Proceedings of the IEEE/ACM 40th International Conference on Software Engineering (ICSE)*. IEEE, 933–944.

[19] Tihomir Gvero and Viktor Kuncak. 2015. Interactive synthesis using free-form queries. In *Proceedings of the IEEE/ACM 37th IEEE International Conference on Software Engineering*. IEEE, 689–692.

[20] 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 International Conference on Software Engineering*. IEEE Press, 842–851.

[21] Emily Hill, Lori Pollock, and K. Vijay-Shanker. 2011. Improving source code search with natural language phrasal representations of method signatures. In *Proceedings of the 26th IEEE/ACM International Conference on Automated Software Engineering*. IEEE Computer Society, 524–527.

[22] Sepp Hochreiter and Jürgen Schmidhuber. 1997. Long short-term memory. *Neural Comput.* 9, 8 (1997), 1735–1780.

[23] Reid Holmes, Rylan Cottrell, Robert J. Walker, and Jorg Denzinger. 2009. The end-to-end use of source code examples: An exploratory study. In *Proceedings of the IEEE International Conference on Software Maintenance*. IEEE, 555–558.

[24] Roya Hosseini and Peter Brusilovsky. 2013. JavaParser: A fine-grain concept indexing tool for Java problems. In *CEUR Workshop Proceedings*, Vol. 1009. University of Pittsburgh, 60–63.

[25] Hamel Husain, Ho-Hsiang Wu, Tiferet Gazit, Miltiadis Allamanis, and Marc Brockschmidt. 2019. CodeSearchNet challenge: Evaluating the state of semantic code search. *arXiv preprint arXiv:1909.09436* (2019).

[26] 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*. 92–101.

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

[28] Iman Keivanloo, Juergen Rilling, and Ying Zou. 2014. Spotting working code examples. In *Proceedings of the 36th International Conference on Software Engineering*. ACM, 664–675.

[29] Jinhan Kim, Sanghoon Lee, Seung-won Hwang, and Sunghun Kim. 2010. Towards an intelligent code search engine. In *Proceedings of the AAAI Conference on Artificial Intelligence*.

[30] Ken Krugler. 2013. Krugle code search architecture. In *Finding Source Code on the Web for Remix and Reuse*. Springer, 103–120.

[31] Xuan Bach D. Le, David Lo, and Claire Le Goues. 2016. History driven program repair. In *Proceedings of the IEEE 23rd International Conference on Software Analysis, Evolution, and Reengineering (SANER)*. IEEE, 213–224.

[32] Claudia Leacock and Martin Chodorow. 1998. Combining local context and WordNet similarity for word sense identification. *WordNet: Electron. Lexic. Datab.* 49, 2 (1998), 265–283.

[33] Otávio Augusto Lazzarini Lemos, Sushil Bajracharya, Joel Ossher, Paulo Cesar Masiero, and Cristina Lopes. 2011. A test-driven approach to code search and its application to the reuse of auxiliary functionality. *Inf. Softw. Technol.* 53, 4 (2011), 294–306.

[34] Otávio Augusto Lazzarini Lemos, Sushil Krishna Bajracharya, Joel Ossher, Ricardo Santos Morla, Paulo Cesar Masiero, Pierre Baldi, and Cristina Videira Lopes. 2007. CodeGenie: Using test-cases to search and reuse source code. In *Proceedings of the 22nd IEEE/ACM International Conference on Automated Software Engineering*. ACM, 525–526.

[35] Hongwei Li, Zhenchang Xing, Xin Peng, and Wenyun Zhao. 2013. What help do developers seek, when and how? In *Proceedings of the 20th Working Conference on Reverse Engineering (WCRE)*. IEEE, 142–151.

[36] Xuan Li, Zerui Wang, Qianxiang Wang, Shoumeng Yan, Tao Xie, and Hong Mei. 2016. Relationship-aware code search for JavaScript frameworks. In *Proceedings of the 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*. ACM, 690–701.

[37] Chao Liu, Dan Yang, Xiaohong Zhang, Haibo Hu, Jed Barson, and Baishakhi Ray. 2018. Poster: A recommender system for developer onboarding. In *Proceedings of the IEEE/ACM 40th International Conference on Software Engineering: Companion (ICSE)*. IEEE, 319–320.

[38] Chao Liu, Dan Yang, Xiaohong Zhang, Baishakhi Ray, and Md Masudur Rahman. 2018. Recommending GitHub projects for developer onboarding. *IEEE Access* 6 (2018), 52082–52094.

[39] Meili Lu, Xiaobing Sun, Shaowei Wang, David Lo, and Yucong Duan. 2015. Query expansion via wordnet for effective code search. In *Proceedings of the IEEE 22nd International Conference on Software Analysis, Evolution, and Reengineering (SANER)*. IEEE, 545–549.

[40] Daniel Lucredio, Antonio Francisco do Prado, and Eduardo Santana de Almeida. 2004. A survey on software components search and retrieval. In *Proceedings of the 30th Euromicro Conference*. IEEE, 152–159.

[41] 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 (E). In *Proceedings of the 30th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 260–270.

[42] Lee Martie and Andre Van der Hoek. 2015. Sameness: An experiment in code search. In *Proceedings of the IEEE/ACM 12th Working Conference on Mining Software Repositories*. IEEE, 76–87.

[43] Frank J. Massey Jr. 1951. The Kolmogorov-Smirnov test for goodness of fit. *J. Amer. Statist. Assoc.* 46, 253 (1951), 68–78.

[44] Collin McMillan, Mark Grechanik, and Denys Poshyvanyk. 2012. Detecting similar software applications. In *Proceedings of the 34th International Conference on Software Engineering*. IEEE Press, 364–374.

[45] Collin McMillan, Mark Grechanik, Denys Poshyvanyk, Qing Xie, and Chen Fu. 2011. Portfolio: Finding relevant functions and their usage. In *Proceedings of the 33rd International Conference on Software Engineering*. ACM, 111–120.

[46] Collin McMillan, Negar Hariri, Denys Poshyvanyk, Jane Cleland-Huang, and Bamshad Mobasher. 2012. Recommending source code for use in rapid software prototypes. In *Proceedings of the 34th International Conference on Software Engineering*. IEEE Press, 848–858.

[47] Ali Mili, Rym Mili, and Roland T. Mittermeir. 1998. A survey of software reuse libraries. *Ann. Softw. Eng.* 5, 1 (1998), 349–414.

[48] Hafedh Mili, Fatma Mili, and Ali Mili. 1995. Reusing software: Issues and research directions. *IEEE Trans. Softw. Eng.* 21, 6 (1995), 528–562.

[49] George A. Miller. 1998. *WordNet: An Electronic Lexical Database*. The MIT Press.

[50] Alon Mishne, Sharon Shoham, and Eran Yahav. 2012. Typestate-based semantic code search over partial programs. In *Proceedings of the ACM International Conference on Object-oriented Programming Systems Languages and Applications*. 997–1016.

[51] Anh Tuan Nguyen, Tung Thanh Nguyen, Hoan Anh Nguyen, Ahmed Tamrawi, Hung Viet Nguyen, Jafar Al-Kofahi, and Tien N. Nguyen. 2012. Graph-based pattern-oriented, context-sensitive source code completion. In *Proceedings of the 34th International Conference on Software Engineering (ICSE)*. IEEE, 69–79.

[52] Lawrence Page, Sergey Brin, Rajeev Motwani, and Terry Winograd. 1999. *The PageRank Citation Ranking: Bringing Order to the Web*. Technical Report. Stanford InfoLab.

[53] 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.

[54] Luca Ponzanelli, Gabriele Bavota, Andrea Mocci, Massimiliano Di Penta, Rocco Oliveto, Mir Hasan, Barbara Russo, Sonia Haiduc, and Michele Lanza. 2016. Too long; didn't watch!: Extracting relevant fragments from software development video tutorials. In *Proceedings of the 38th International Conference on Software Engineering*. ACM, 261–272.

[55] Mukund Raghothaman, Yi Wei, and Youssef Hamadi. 2016. SWIM: Synthesizing what I mean-code search and idiomatic snippet synthesis. In *Proceedings of the IEEE/ACM 38th International Conference on Software Engineering (ICSE)*. IEEE, 357–367.

[56] Md Masudur Rahman, Jed Barson, Sydney Paul, Joshua Kayani, Federico Andrés Lois, Sebastián Fernandez Quezada, Christopher Parnin, Kathryn T. Stolee, and Baishakhi Ray. 2018. Evaluating how developers use general-purpose web-search for code retrieval. In *Proceedings of the 15th International Conference on Mining Software Repositories*. ACM, 465–475.

[57] Veselin Raychev, Martin Vechev, and Eran Yahav. 2014. Code completion with statistical language models. In *ACM Sigplan Notices*, Vol. 49. ACM, 419–428.

[58] Steven P. Reiss. 2009. Semantics-based code search. In *Proceedings of the 31st International Conference on Software Engineering*. IEEE Computer Society, 243–253.

[59] Romain Robbes and Michele Lanza. 2010. Improving code completion with program history. *Autom. Softw. Eng.* 17, 2 (2010), 181–212.

[60] Martin Robillard, Robert Walker, and Thomas Zimmermann. 2009. Recommendation systems for software engineering. *IEEE Softw.* 27, 4 (2009), 80–86.

[61] Caitlin Sadowski, Kathryn T. Stolee, and Sebastian Elbaum. 2015. How developers search for code: A case study. In *Proceedings of the 10th Joint Meeting on Foundations of Software Engineering*. ACM, 191–201.

[62] Gerard Salton, Edward A. Fox, and Harry Wu. 1983. Extended Boolean information retrieval. *Commun. ACM* 26, 11 (1983), 1022–1036.

[63] Jianhang Shuai, Ling Xu, Chao Liu, Meng Yan, Xin Xia, and Yan Lei. 2020. Improving code search with co-attentive representation learning. In *Proceedings of the 28th International Conference on Program Comprehension*. pages 196–207.

[64] Susan Elliott Sim, Medha Umarji, Sukanya Ratanotayanon, and Cristina V. Lopes. 2011. How well do search engines support code retrieval on the web? *ACM Trans. Softw. Eng. Methodol.* 21, 1 (2011), 4.

[65] Janice Singer, Timothy Lethbridge, Norman Vinson, and Nicolas Anquetil. 2010. An examination of software engineering work practices. In *CASCON First Decade High Impact Papers*. IBM Corp., 174–188.

[66] Kathryn T. Stolee, Sebastian Elbaum, and Daniel Dobos. 2014. Solving the search for source code. *ACM Trans. Softw. Eng. Methodol.* 23, 3 (2014), 26.

[67] Anthony J. Viera, Joanne M. Garrett, et al. 2005. Understanding interobserver agreement: The kappa statistic. *Fam. Med.* 37, 5 (2005), 360–363.

[68] Yao Wan, Jingdong Shu, Yulei Sui, Guandong Xu, Zhou Zhao, Jian Wu, and Philip S. Yu. 2019. Multi-modal attention network learning for semantic source code retrieval. *arXiv preprint arXiv:1909.13516* (2019).

[69] Frank Wilcoxon. 1945. Individual comparisons by ranking methods. *Biomet. Bull.* 1, 6 (1945), 80–83.

[70] Xin Xia, Lingfeng Bao, David Lo, Pavneet Singh Kochhar, Ahmed E. Hassan, and Zhenchang Xing. 2017. What do developers search for on the web? *Empir. Softw. Eng.* 22, 6 (2017), 3149–3185.

[71] Tao Xie and Jian Pei. 2006. MAPO: Mining API usages from open source repositories. In *Proceedings of the International Workshop on Mining Software Repositories.* 54–57.

[72] Feng Zhang, Haoran Niu, Iman Keivanloo, and Ying Zou. 2017. Expanding queries for code search using semantically related API class-names. *IEEE Trans. Softw. Eng.* 44, 11 (2017), 1070–1082.

[73] Neng Zhang, Jian Wang, and Yutao Ma. 2017. Mining domain knowledge on service goals from textual service descriptions. *IEEE Trans. Serv. Comput.* 13, 3 (2017), 488–502.

[74] Yun Zhang, David Lo, Pavneet Singh Kochhar, Xin Xia, Quanlai Li, and Jianling Sun. 2017. Detecting similar repositories on GitHub. In *Proceedings of the IEEE 24th International Conference on Software Analysis, Evolution and Reengineering (SANER).* IEEE, 13–23.

Received August 2020; revised March 2021; accepted May 2021