Are the Code Snippets What We Are Searching for?

A Benchmark and An Empirical Study on Code Search with Natural-Language Queries

Shuhan Yan, Hang Yu, Yuting Chen*, and Beijun Shen*
Shanghai Jiao Tong University, Shanghai, China.
{yansh0625, yuhang012, bjshen, chenyt}@sjtu.edu.cn

Lingxiao Jiang
Singapore Management University, Singapore.
lxjiang@smu.edu.sg

***Abstract*—Code search methods, especially those that allow programmers to raise queries in a natural language, plays an important role in software development. It helps to improve programmers' productivity by returning sample code snippets from the Internet and/or source-code repositories for their natural-language queries. Meanwhile, there are many code search methods in the literature that support natural-language queries. Difficulties exist in recognizing the strengths and weaknesses of each method and choosing the right one for different usage scenarios, because (1) the implementations of those methods and the datasets for evaluating them are usually not publicly available, and (2) some methods leverage different training datasets or auxiliary data sources and thus their effectiveness cannot be fairly measured and may be negatively affected in practical uses.**

To build a common ground for measuring code search methods, this paper builds CosBench, a dataset that consists of 1000 projects, 52 code-independent natural-language queries with ground truths, and a set of scripts for calculating four metrics on code research results. We have evaluated four *IR (Information Retrieval)-based* and two *DL (Deep Learning)-based* code search methods on CosBench. The empirical evaluation results clearly show the usefulness of the CosBench dataset and various strengths of each code search method. We found that DL-based methods are more suitable for queries on *reusing code*, and IR-based ones for queries on *resolving bugs* and *learning API uses*.

***Index Terms*—natural-language code search, benchmarking, empirical study, information retrieval, machine learning, deep learning, word embedding**

I. Introduction

Code search plays an important role in software development [22], [39], [40], [43], [53]. In particular, code search methods that allow programmers to raise queries in natural languages (abbreviated as *natural-language code search* in this paper) are more convenient for programmers to use than those that need specific query languages. In a natural language, programmers can describe their needs for implementing specific algorithms and/or functionalities, finding code samples that use specific APIs, or seeking for code solutions to hard problems, and then natural-language code search methods can retrieve code snippets meeting the needs from the Internet or code repositories. Many natural-language code search methods have been proposed in the literature, some of which are available as either open-source or commercial tools [2], [4], [6].

*Beijun Shen and Yuting Chen are corresponding coauthors.

The existing natural-language code search methods can be classified into two mainstreams [8]: *IR (Information Retrieval)-based* methods and *DL (Deep Learning)-based* ones. The two kinds of search methods differ in their respective styles of matching queries and code snippets. An IR-based method usually extracts from a query a set of keywords and then searches for the keywords in code repositories [24], [25], [31], [45], [51]. Comparatively, a DL-based method takes some deep learning techniques, especially embedding algorithm(s), that map raw data (including queries and code snippets) into a high-dimensional space and then matches them [13], [49].

Few studies have evaluated the effectiveness of various code search methods against each other extensively, although many papers exist in proposing different search techniques. Indeed, difficulties exist in empirically evaluating code search methods fairly and extensively against each other.

Difficulty 1. Lack of a common dataset and consistent evaluation metrics.

Many studies on code search (*e.g.*, [13], [25], [31], [49], [52]) collected and used their own datasets, and used different metrics in their evaluations. Such inconsistencies in datasets and metrics make it difficult to compare the evaluation results across studies fairly. For a fair comparison, a dataset, including a common codebase for search and a common set of queries, should be carefully curated, and the evaluation metrics should be consistently chosen.

Difficulty 2. Too many peripheral factors that may affect code search results when comparing different implementations of code search methods.

Many code search methods, especially commercial code search engines, only provide query interfaces to search within their backends. Their implementations and datasets are not publicly available, and it is difficulty to check if their effectiveness is really attributed to their search techniques or some other factors. For example, some methods may leverage special training datasets or auxiliary data sources to enhance themselves; a codebase may be differently preprocessed and indexed; queries may be differently preprocessed or expanded, *etc.* Consequently, existing studies often resort to a comparison against some rudimentary open-sourced methods (*e.g.*, Lucene [5]) or an older version of their own work, so that the evaluation can actually focus on evaluating the effectiveness of the core parts of the IR/DL algorithms in their methods, excluding the effects of other peripheral factors.

In order to gain insights into the strength and weakness of each code search method, a fair comparison should also be able to evaluate chosen components of a method, in addition to using a common dataset and consistent metrics.

Difficulty 3. Unclear intentions expressed in the queries from programmers. Natural-language queries are often informally presented, and their literal meaning may not reflect programmers' real needs (*e.g.*, programmers may unknowingly use a wrong word in a query; programmers may not be able to tell their purposes in querying for certain code.). Different intentions in queries may or may not be processed by different code search methods and affect their effectiveness. Mixing queries of different types of intentions in evaluations may obscure the effectiveness of different search methods. For a more insightful comparison of the strengths and weaknesses of different code search methods, the evaluation data should also include queries of different types of intentions.

Essentially, this paper aims to address the following key research question while overcoming these evaluation difficulties:

Question: "*Which is the best method for which type of queries among the existing natural-language code search methods?*"

Towards answering the above question, this paper aims to build a common ground for fair and comprehensive evaluation of natural-language code search methods, and investigate, given a query of a specific type of intention, which code search method returns the most relevant results. More specifically, this paper makes the following contributions.

Dataset. We have built CosBench, a dataset that can enable fair and extensive evaluation of natural-language code search methods. It currently contains 1000 Java projects collected from GitHub and 52 queries with ground truths of three types of intentions (*i.e.*, bug resolution, code reuse, and API learning). It also contains scripts to calculate four metrics (Precision@k, Mean Average Precision@k, Mean Reciprocal Rank@k, and Frank measures) for evaluating code search results. CosBench is publicly available on GitHub.[1]

Implementation. With CosBench, we have also implemented four representative code search methods that take natural-language queries as input, including three IR-based and a DL-based method (cf. Section III), and included two publicly available code search methods Lucene and Code-nn [13]. These methods are implemented in a unified, Lucene-compatible framework, allowing the implementations to be compared fairly.

Evaluation. We have evaluated the six code search methods against each other on CosBench. The empirical results clearly show the usefulness of the CosBench dataset and the strength of each natural-language code search method. In particular, we found that the DL-based code search methods are more suitable for queries on reusing code, while the IR-based ones for queries on resolving bugs and learning API uses.

The rest of the paper is organized as follows. Section II presents the CosBench dataset. Section III describes the code search methods chosen for evaluation. Section IV evaluates the code search methods on the dataset. Section V discusses various threats to validity. Section VI compares with related work. Section VII concludes.

[1] https://github.com/BASE-LAB-SJTU/

II. CosBench Dataset

This section presents the CosBench dataset used for evaluating code search methods that take natural-language queries as input. CosBench has three components:

$$\textsf{CosBench} = Codebase + QASet + Metrics$$

A. Codebase

The codebase of CosBench consists of 1,000 Java projects, with $475,783$ Java files and $4,199,769$ code snippets (*i.e.*, Java methods). The total size of the codebase is 1.4G. Table I enumerates the top 10 most popular projects in the codebase.

The code snippets are all collected from GitHub. Firstly, we sorted all Java projects on GitHub by their popularities (denoted by the stars) and selected the top 1,000 most popular ones. Secondly, we extracted the Java source files (.java) from these projects and divided them into code snippets by following the technique in [13]. Each code snippet is a Java method, and comments, line breaks, and redundant spaces are removed.

B. QASet (Query and Answer Set)

1) Queries and Their Types: CosBench contains a set of codebase-independent queries. These queries are categorized into either *phrase* or *keyword* queries. A phrase query is usually raised as a sentence or a phrase, *e.g.*, "`How to convert an image to base64 encoding?`" A keyword query often contains one or more keywords that need to be strictly matched with code snippets, *e.g.* "`image encoding base64`".

Each query also has its intention, representing the user's expectation(s) and/or potential usages of the searching results. Some studies have summarized the common intentions of queries frequently raised by programmers [8], [37], [38], [48]:

(a) *Code reuse*. Queries can be raised for reusing code to avoid repetitive implementations or to find the best industrial practices. A query for reusing code often contains some functional descriptions;
(b) *API learning*. Some queries may be raised for learning how to use APIs. Such a query often contains API names;
(c) *Bug resolution*. Many programmers may raise queries for resolving program bugs. Such a query is usually long, with a bug report;
(d) *Traceability*. Programmers may search for locations of certain functions and/or code snippets in software projects;
(e) *Programming knowledge*. Programmers may search for programming knowledge, such as guidelines of using a new language, coding conventions, design patterns, *etc.*;
(f) *Domain knowledge*. Programmers may search for domain knowledge, *e.g.*, machine learning, image processing, *etc.*;
(g) *Tool uses*. Programmers may search for code related to tools, *e.g.*, IDEs, version control tools, tool configurations, *etc.*;

TABLE I
TOP 10 MOST POPULAR PROJECTS IN THE CODEBASE. THE PROJECTS ARE SORTED IN DESCENDING ORDER OF THEIR POPULARITIES.

Project	Project Description	#Files	#Methods	Size	Stars
java-design -patterns	Design patterns implemented in Java.	1,126	2,841	959KB	51.8k
elasticsearch	Distributed, RESTful search engine.	11,081	89,723	60MB	44.6k
spring-boot	Spring-powered, production-grade applications and services.	4,636	49,320	21MB	42.3k
RxJava	Reactive extensions for the JVM.	1,652	29,700	13MB	40.7k
okhttp	An HTTP+HTTP/2 client for Android and Java applications.	189	10,981	6.1MB	34.6k
guava	Google core libraries for Java.	3,145	59,789	25M	34.1k
retrofit	Type-safe HTTP client for Android and Java by Square, Inc.	241	2,115	1,012KB	33.9k
spring-framework	The framework for all Spring projects.	7,100	110,023	52MB	32.7k
dubbo	A high-performance, Java based RPC framework.	1,691	9,877	4.2MB	29.4k
MPAndroidChart	A powerful & easy to use chart library forAndroid.	220	2,115	1.0MB	28.7k

TABLE II
NUMBERS OF DIFFERENT TYPES OF QUERIES AND SOME EXAMPLES.

	Code Reuse	API Learning	Bug Resolution	Total
Phrase/Keyword	23	14	15	52

Sample queries. **#SO** and **#CS** show the numbers of relevant answers curated from StackOverflow posts and the results of existing code search methods.

Query	Intention	Rep.	#SO	#CS
Q1 `How do I invoke a Java method when given the method name as a string?`	*Reuse*	*Phrase*	6	35
Q2 `invoke method by method name`	*Reuse*	*Keyword*	6	35
Q3 `Can I use Class.newInstance() with constructor arguments?`	*API*	*Phrase*	3	7
Q4 `Class.newInstance() constructor arguments`	*API*	*Keyword*	3	7
Q5 `How to fix Illegal-MonitorStateException? my code is try{page.wait(1);}`	*Bug*	*Phrase*	1	2
Q6 `IllegalMonitorState-Exception`	*Bug*	*Keyword*	1	2

(h) *Others*. Other query intentions include how to connect to a database, how to write test scripts, *etc.*

We have observed that the first three intentions above have the most numbers of queries on StackOverflow. CosBench thus currently collects only queries of one of the three query intentions: *reusing*, *API learning* and *bug resolution*. Table II shows an overview of the queries chosen in CosBench and some examples. Note that the query intentions are identified by human engineers, on the basis of their descriptions and characteristics (including API names, bug descriptions, *etc.*).

We curated 52 queries by selectively choosing posts from Stack Overflow (SO) [7]. Firstly, we investigated the list of Java-tagged questions posted on SO and sorted them by their vote numbers—A Java-tagged post with a sufficient number of votes usually contains Java code as relevant results. Secondly, we picked 26 posts from the Top-50 posts(*i.e.*, those shown on the first page); these posts were manually chosen as they are much more relevant to the three intentions we are studying. The titles of the posts were taken as phrase queries. Keywords, which were manually extracted from these phrase queries, were taken as keyword queries in the dataset.

2) Query Answers: The ground truths (*i.e.*, the relevant answers) to the 52 queries are also curated and included in the dataset. For example, an answer to the query "`How to make pipes work with Runtime.exec()?`" is:

TABLE III
SIZES OF QUERY ANSWERS.

Mean # of words per answer	**Code Reuse**	**API Learning**	**Bug Resolution**	**All**
	18.61	26.56	27.18	22.78

```
String[] cmd = {
"/bin/sh",
"-c",
"ls /etc | grep release"
};

Process p = Runtime.getRuntime().exec(cmd);
```

We curated the ground truths in two respects.

(i) SO answers. We extracted the code snippets from the SO posts that are marked as answers. We manually checked the correctness of these code snippets and included the correct ones in the dataset.

(ii) Code search results. To curate more possibly correct answers for each query, we ran all the chosen code search methods (see Section IV) on the CosBench codebase and collected their results returned for each query, analyzed the results manually, and then added the relevant ones into our dataset.

Note that the answer set for each query may still be incomplete or inaccurate, as we do not have a prophet that can retrieve a complete and precise set of true answers. Nevertheless, we manually vetted through the possible answers collected from the two respects for each query to build the answer set, so that the relative precision and recall of each code search method can be estimated.

Each query in CosBench has, on average, 3.56 SO answers and 13.14 code search results. Table II shows the numbers of relevant SO answers and code search results for sample queries. The answers to a query can be long or short. Meanwhile, as shown in Table III, we note that the answers to the queries on code reusing are on average shorter than those on API learning and resolving bugs.

C. Metrics

We surveyed the metrics used in several studies [9], [10], [13], [18], [19], [24], [25], [31], [33], [35], [45], [46], [49], [51], [52]. Metrics, including *Precision*,*Recall*,*MAP*,*MRR*, *F-Score*, *NDCG*, and *Frank*, are often used in previous studies; different researchers may choose to use different metrics in their evaluations.

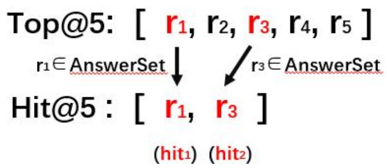


Fig. 1. An example of *Top@k* and *Hit@k*.

CosBench provides scripts to calculate four of the metrics: *Precision@k*, *MAP@k*, *MRR@k*, and *Frank*; it leaves the other metrics (*e.g.*, *Recall*, *F-score*, and *NDCG*) for future work when we include more relevant answers for each query and have more confidence in the completeness of the answer set for calculating recalls.

Let $Q = [q_1, q_2, \ldots, q_n]$ be the set of queries to be performed. Let *AnswerSet* be the ground-truth set of answers *w.r.t.* a query q. As Figure 1 shows, let $Top@k = [r_1, r_2, \ldots, r_k]$ and $Hit@k = [hit_1, hit_2, \ldots, hit_m]$ be a sequence of the top-k results retrieved and a projection of *AnswerSet* onto *Top@k*, respectively.

The four metrics in current CosBench are then defined.

(1) *Precision@k* measures how many ground-truth answers are hit on average in the *Top@k* returned for a query in Q:

$$Precision@k = \frac{1}{|Q|} \sum_{i=1}^{|Q|} \frac{|Hit@k(q_i)|}{k} \tag{1}$$

Precision@k shows the relevance of the returned results to the queries with respect to the ground-truth answers. The higher the value, the more relevant the results are.

(2) *MAP@k* is the mean average precision across the rankings returned for all the queries:

$$MAP@k = \frac{1}{|Q|} \sum_{i=1}^{|Q|} \frac{1}{m} \sum_{j=1}^{m} \frac{j}{rank(hit_j, Top@k(q_i))} \tag{2}$$

where m is $|Hit@k|$, hit_i is an element in $Hit@k$, and $rank(e,l)$ is the rank (*i.e.*, the index) of an element e in a list l. When $|Hit@k(q_i)| = 0$, hit_j does not exist and the average precision for q_i is set 0. The higher the *MAP@k* value is, the more answers are hit by the top-k results.

(3) *MRR@k* is the mean reciprocal rank across all queries:

$$MRR@k = \frac{1}{|Q|} \sum_{i=1}^{|Q|} \frac{1}{rank(hit_1, Top@k(q_i))} \tag{3}$$

When $|Hit@k(q_i)| = 0$, the reciprocal rank is set to 0. Usually only the first hit is considered. The higher the *MRR@k* value is, the higher ranked the hit answers are in *Top@k*.

(4) *Frank@k* is the mean rank of the first hit answer across all queries [33]:

$$Frank@k = \frac{1}{|Q|} \sum_{i=1}^{|Q|} Frank@k(q_i) \tag{4}$$

where $Frank@k(q_i) = rank(hit_1(q_i), Top@k(q_i))$. Clearly, the smaller the *Frank@k* value is, the earlier ground-truth answers appear in the search results.

TABLE IV
Papers related with natural-language code searches. Accessed December, 2019.

	ASE	TOSEM	ICSE	ICDL	ICSME	SANER	other	all
2014	1	1	1		1			4
2015	2			1		1	1	5
2016			2				2	4
2017			1		1		1	3
2018			2				1	3
2019			1				1	2
all	3	1	7	1	2	1	6	21

In the evaluation, we let k be 3 or 20, as programmers may often have the patience to examine top-3 results, while there may be often enough space to display top-20 results on the first page for search results.

III. Code Search Methods

A code search engine (*i.e.*, an implementation of a code search method), is typically composed of various components that parse queries in natural languages, process code snippets, and match code snippets with queries. Each component can take some different strategies, incurring extra efforts in implementing and integrating these strategies into one engine.

We selected six state-of-the-art code search methods in our empirical study.

A. Selection of Code Search Methods

We did a survey on natural-language code search and took three steps to select code search methods for evaluation.

Firstly, we searched Google Scholar [3], looking for the papers that were accepted by some top venues in 2015-2018. The keywords used for searches were "`code search|recommend code|code recommender|code queries|code snippet|query code`". The conferences were restricted to five closely related ones, *i.e.*, FSE, ICSE, ASE, ICSME, and SANER. In this step we got 46 papers.

Secondly, we investigated the 46 papers and their relevance to the area of code search. We also went through the references of these papers. In this step, 21 papers, which are shown in Table IV, were identified to be strongly related to this topic of code search for natural-language queries.

Thirdly, we used the following criteria for choosing code search methods for evaluation: ① "Is the method suitable for Java?", ② "Is the search process fully automated?", ③ "Is the implementation or the dataset publicly available?", ④ "Can the method be implemented?", ⑤ "Is the method representative?"—A method is "representative" if it takes strategies significantly different from those taken by the methods we have chosen.

We finally selected six code search methods, including Lucene [5], LuSearch [24], CodeHow [25], QECK [31], YeSearch [49], and Code-nn [13]. Among them, only Lucene and Code-nn are publicly available. We have re-implemented CodeHow, YeSearch, LuSearch, and QECK, following what are described in their paper faithfully, to facilitate fair comparison against Lucene and Code-nn on the CosBench dataset.

B. IR-based Methods

The essence of the IR-based methods is to search for the keywords/words of a query in the codebase [17]. Four of the six methods we chose, *i.e.*, Lucene [5], LuSearch [24], CodeHow [25], and QECK [31], can be classified as IR-based methods. Lucene is a commonly used framework for any text search.

LuSearch, CodeHow, and QECK follow a Lucene-compatible process to search code. As Figure 2(a) shows, the process is mainly composed of three steps:

- *Preprocessing*. This step preprocesses code snippets and queries by removing the stopwords, splitting compound words, lowering cases and stemming. Here we keep the preprocessing step the same for the four methods such that the effects of their strategies for information enhancement and similarity calculation can be compared fairly.
- *Information enhancement*. As user queries can be codebase independent and use words or abbreviations different from code, this step expands queries in various ways such that synonyms of query words can be also used for code searches.
- *Similarity calculation*. This step calculates the similarities between queries and code snippets to return search results.

We use Lucene version 7.4.0 [5] as the rendering framework. The main differences among the four IR-based methods are in *information enhancement* and *similarity calculation* steps, as shown in the boxes with red dashed boundaries in Figure 2(a).

C. DL-based Methods

DL-based methods advocate the idea of mapping and matching data in a high-dimensional numerical vector space. Word and code embedding algorithms based on neural networks are employed to learn the mapping and/or matching rules from historical data used for training [20].

Two DL-based methods are chosen for CosBench.

(1) YeSearch [49]. YeSearch proposed by Ye *et al.* bridges the vocabulary gap between natural-language queries and code snippets by projecting them into the same vector space.
(2) Code-nn [13]. Code-nn proposed by Gu *et al.* embeds code snippets and their natural-language descriptions into a vector space, making code snippets and their descriptions comparable to queries.

As Figure 2(b) shows, YeSearch and Code-nn differ in four aspects: ① features extracted, ② word embeddings, ③ semantic representations of the corpus, and ④ similarity calculation algorithms taken.

IV. Evaluation

The evaluation is designed to evaluate the usefulness of the CosBench dataset with respect to the key research question raised in Section I. In particular, we use CosBench to answer the two specific questions:

RQ1 (Strength): Which is the best method among the existing natural-language code search methods?

RQ2 (Query Type): How do the types of queries and intentions affect the search results?

TABLE V
NUMBERS OF QUERIES THAT CAN BE ANSWERED BY THE CODE SEARCH METHODS. A QUERY IS "ANSWERED" IF $|Hit@20| > 0$ HOLDS.

# of queries	Lucene	LuSearch	CodeHow	QECK	YeSearch	Code-nn
answered	39	13	35	32	33	29

TABLE VI
$|Hit@20|$ *w.r.t.* THE EXAMPLE QUERIES IN TABLE II

Query	Lucene	LuSearch	CodeHow	QECK	YeSearch	Code-nn
Q1	11	0	13	2	9	9
Q2	11	0	13	2	9	17
Q3	1	0	3	5	2	6
Q4	3	1	5	3	4	6
Q5	1	0	2	0	1	0
Q6	1	0	2	0	1	0

A. Usefulness of CosBench

CosBench is sufficient for measuring natural-language code search methods: The codebase contains a rich set of code snippets, the natural-language queries and their ground-truth answers that are manually vetted, and four evaluation metrics. In particular, each query has $2 \sim 42$ (16.7 on average) answers; the effectiveness of a code search method can then be demonstrated through investigating the results retrieved and the answers hit.

As Table V shows, each code search method can answer $13 \sim 39$, but not all, of the 52 queries in their $Top@20$ results.

Table VI shows that the six code search methods perform differently with respect to the CosBench dataset, at least the sample queries in Table II. Lucene, CodeHow, and YeSearch can answer all of the six queries, while LuSearch only one. A detailed explanation about the effectiveness of each code search method is given in Section IV-B.

The above results lead to our first observation:

Observation 1. *CosBench is useful for evaluating natural-language code searches in that (1) it contains a rich set of code snippets and a diverse query-answer set; (2) all of the six method can be evaluated and their capabilities be demonstrated and differentiated by the dataset.*

B. Results to RQ1

We evaluated the six methods on the dataset. The average results are shown in Table VII, and box plots of the performance measures of the methods are shown in Figure 3.

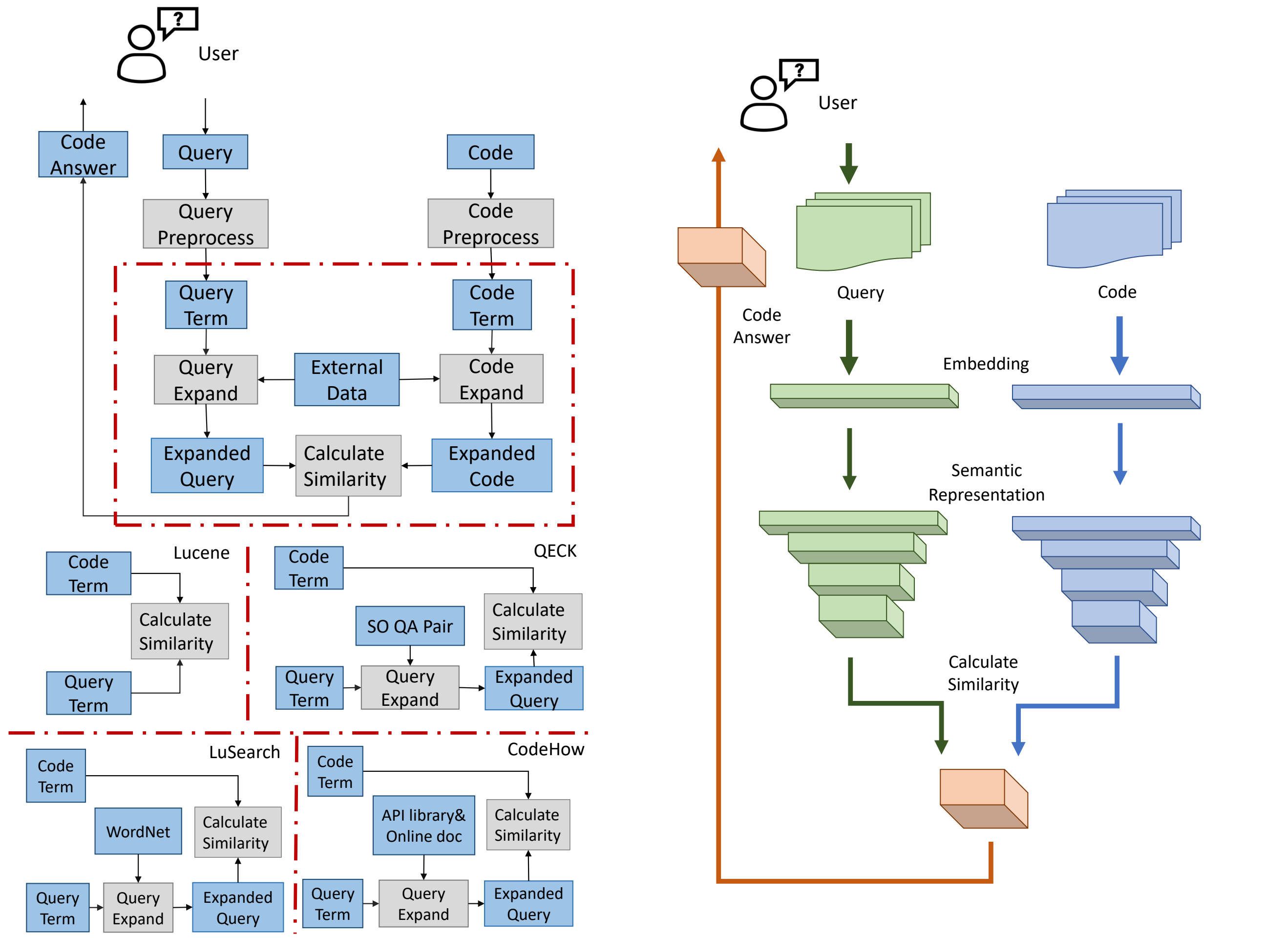
First, Code-nn achieves the highest $Precision@3$ value (0.2115) among the six search methods. Meanwhile, along with the increase of k, $Precision@k$ of the IR-based methods can increase more rapidly than that of Code-nn. For instance, Lucene and CodeHow have higher $Precision@20$ values (0.2916 and 0.3090) than Code-nn (0.2229).

Second, Code-nn outperforms the others on $MAP@3$—it indicates that Code-nn hits more answers in its $Top@3$ than the others, or its hit answers are of higher ranks than the answers hit by the others. Meanwhile, Lucene and CodeHow

[2] https://wordnet.princeton.edu/download/current-version
[3] https://docs.oracle.com/javase/8/docs/api/
[4] https://archive.org/details/stackexchange

Method	Query Expand	Similarity
Lucene	/	BM25 [1]
LuSearch [24]	WordNet 3.0[2]	Equation based on Co-occurrence words [24]
CodeHow [25]	API information from the Java 8 document[3]	A Boolean model [25].
QECK [31]	QA pairs from the Stack Exchange Data Dump [4]	BM25

Method	Feature	Word Embedding	Semantic Representation	Similarity
YeSearch [49]	Token	Skipgram [29]	/	text-to-text [28]
Code-nn [13]	function name, API sequence, token	Mappings of the words to numbers in a dictionary	LSTM [14], [44] for function name, API sequence and query, and MLP [36] for tokens	cosine



(a) Workflow of IR-based methods. A blue box indicates data and a gray box indicates an operation.

(b) Workflow of DL-based methods. Embeddings of natural-language queries (green) are matched with embeddings of code (blue) to generate results (brown).

Fig. 2. An overview of the methods chosen in our study.

TABLE VII
AVERAGE RESULTS ON *Precison@k*, *MAP@k*, *MRR@k* AND *Frank@k*.

Type	Method	P@3	MAP@3	MRR@3	P@20	MAP@20	MRR@20	Frank@20
IR	Lucene	0.1730	0.0314	0.2532	0.2916	0.0972	**0.3237**	**3.667**
	LuSearch	0.0384	0.0046	0.0673	0.0848	0.0284	0.0883	5.615
	CodeHow	0.1538	0.0288	0.2083	**0.3090**	**0.1197**	0.2882	3.971
	QECK	0.1346	0.0202	0.1378	0.1906	0.0731	0.2061	5.25
DL	YeSearch	0.0641	0.0050	0.0801	0.1667	0.0384	0.1551	5.142
	Code-nn	**0.2115**	**0.0320**	**0.2756**	0.2229	0.0459	0.3211	4.034

have higher *MAP*@20 values (0.0972 for Lucene and 0.1197 for CodeHow) than Code-nn (0.0459), indicating that Lucene and CodeHow can hit more answers in their *Top*@20 than Code-nn, or their hit answers are of higher ranks than the answers hit by Code-nn.

Third, the first answer hit by Code-nn or Lucene are of higher ranks than those hit by the others, as Code-nn and Lucene achieve higher *MRR*@3 values (0.2756 and 0.2532) and higher *MRR*@20 values (0.3211 and 0.3237) than the others.

In addition, the *Frank*@20 values of all methods are between 3.667 and 5.147, implying that an answer can on average be found in the first three to five search results.

The following observation summarizes the above results:

Observation 2. *Code-nn achieves the highest Precision@3, MAP@3, MRR@3 values among the six methods; the two DL-based methods achieve lower Precision@20, MAP@20, MRR@20 values than the four IR-based methods.*

1) Analyzing IR-based methods: We took a further analysis of the four IR-based methods in two respects: (1) information

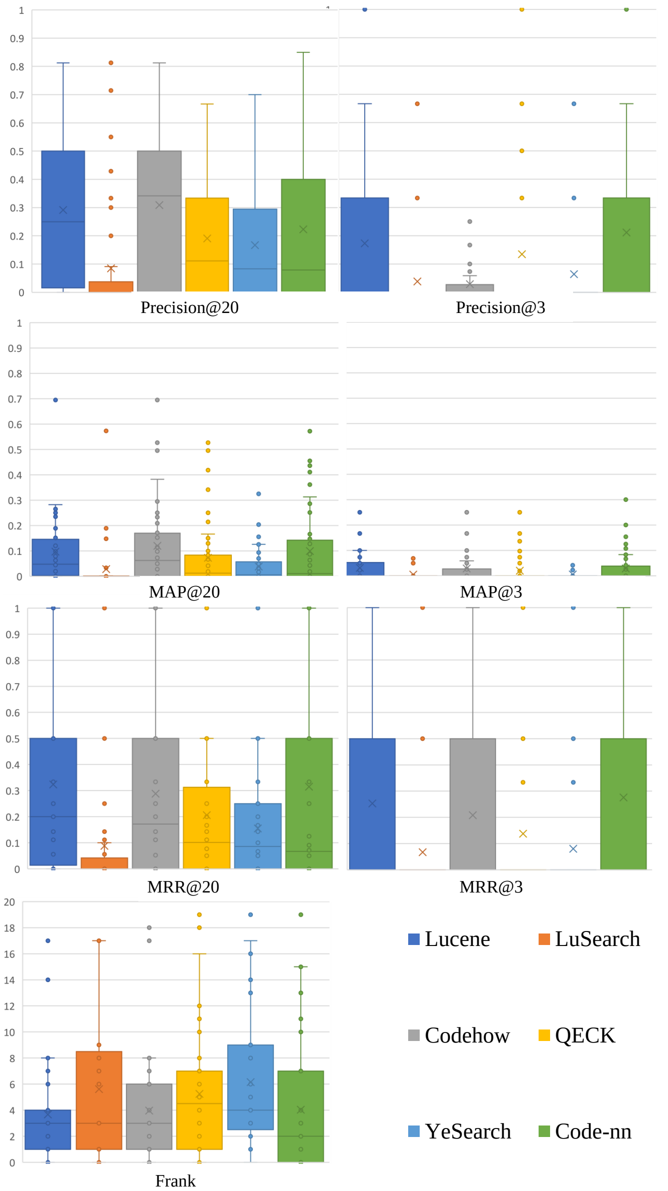
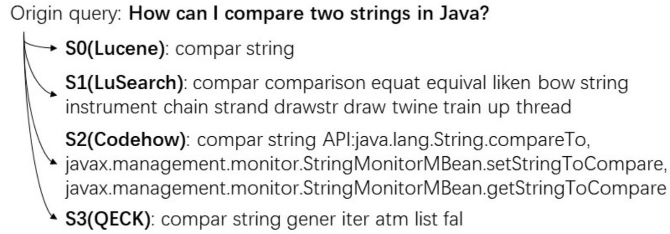


Fig. 3. Box plots of the performance measures of the six methods on *Precison@k*, *MAP@k*, *MRR@k* and *Frank@k*.



Method	P@20	MAP@20	MRR@20
Lucene (S0)	0.55	0.12	0.25
LuSearch (S1)	0	0	0
CodeHow (S2)	0.66	0.30	0.25
QECK (S3)	0.11	0.01	0.16

Fig. 4. A sample query and its expansions, and corresponding search performance measures (when BM25 is employed for retrieval).

enhancement conducted and (2) code snippets returned.

The IR-based methods, except Lucene, expand queries for code searches. Meanwhile, a query expansion may not definitely enhance a code search method, since each method takes its specific dictionary to expand queries. Figure 4 shows a sample query and the expanded queries obtained by the three IR-based methods. Recall from Figure 2(a) that LuSearch, CodeHow, and QECK expand queries using WordNet, APIs in Java native libraries, and QAs from SO, respectively.

We then applied one similarity calculation algorithm (BM25) on the expanded queried to return search results. As shown in Figure 4, different query expansions lead to different results. First, LuSearch's expansion is less effective for code searches, since WordNet provides a synonym dictionary that is not specific for code searches. Second, CodeHow uses APIs in Java native libraries to expand queries. The expanded APIs bridge the gap between queries and code, and thus CodeHow achieves higher *Precision*@20 and *MAP*@20 values than Lucene. However, many native and industrial libraries are used in the codebase, while it remains a problem which libraries should be included for expanding queries. Third, QECK employs QAs from SO to expand queries. These QAs often contain code snippets and thus are much more relevant to the codebase. However, a large number of irrelevant words can be introduced, making the effectiveness of QECK unsatisfying.

The above results lead to the next observation.

Observation 3. *Code-specific information enhancement helps improve code searches.*

2) Comparing Code-nn and YeSearch: We also compared the two DL-based methods, and found that Code-nn is $1.2 \sim 26\times$ higher than YeSearch in its $Precision@k, MAP@k, MRR@k$ values. There are two main reasons for this.

First, YeSearch takes a similarity calculation algorithm that is not suitable for code searches—the average word similarity weighted by IDF (Inverse Document Frequency) [42] is taken to represent the similarity between two sentences; the similarity is easily affected by the similarity between the words in a code snippet and those in a query. As a result, irrelevant code snippets with relevant words are more easily retrieved.

Second, the two DL-based methods use their specific code features: Code-nn uses three features, while YeSearch uses only one, leading to loss of code information. The more code features are employed, the richer the code information is, and the more accuracy a matching is likely to be.

Thus we draw the next observation.

Observation 4. *Code-nn outperforms YeSearch on the CosBench dataset.*

3) Analyzing the time cost: The time cost of an IR-based method includes the time for indexing and searching. The time cost of a DL-based method includes the time for model training and searching. In our study, all methods were run on the Linux operating system (CPU XEON E5-2620 6 core 12 threads 2.4Ghz, 64G DDR4 memory, and one GTX 1080Ti GPU). The models were trained on the GPU. All of the other

TABLE VIII
TIME COST.

Method	Indexing (s)	Training	Each search
Lucene	77.11	/	80.25 ms
CodeHow	250.61	/	3119.25 ms
LuSearch	137.54	/	176.28 ms
QECK	315.44	/	338.5 ms
YeSearch	/	4.75h/50epoch	timeout*
Code-nn	/	227.5h/2000epoch	18.3 ms

* YeSearch is very slow on middle/large-scale codebases.

operations were performed on the CPU. Table VIII describes the time cost of each search method.

The IR-base code searches do not need an offline training process. The four IR-based methods are fast in indexing code snippets. However, CodeHow is slower in code searches than the other IR-based methods, because its similarity calculation is a little more complicated (and inefficient) than those taken by the others.

DL-based method needs longer time to train a model, and updating the model using new data may not be efficient either. On the other hand, the training process is offline. Having the model been trained, the model can be directly used for code search. Code-nn is much faster in code search than YeSearch. In a code search, Code-nn converts a query to a vector, and searches for the k-nearest code vectors in the same vector space. Its time complexity is $O(n)$, where n is the number of code snippets in the codebase. Comparatively, YeSearch calculates the text-to-text similarity. The time complexity is thus $O(2 \times w_c \times w_q \times n)$, where w_c is the number of query words, and w_q the number of words in a code snippet. YeSearch needs a more efficient similarity calculation algorithm when running on a large codebase.

Observation 5. *The IR-based methods are fast in code indexing and search. The DL-based methods are slow in training, but can be fast in code search too; however, YeSearch is much slower than Code-nn in code search.*

C. Results to RQ2

We looked further into the queries in the CosBench dataset and their ground-truth answers, and found that:

(1) For the queries on *reusing code*, the queries and the function names of the answers are strongly related. In particular, 61.32% of answers have function names semantically related to the queries.

(2) For the queries on *resolving bugs*, the buggy code raised in the queries and the answers are syntactically similar. Thus clone detection may help reveal answers to such queries. In particular, for queries containing specific exceptions (*e.g.*, `IllegalMonitorStateException`), matching exception names may facilitate code identification.

(3) For the queries on *using APIs*, API names are included in most answers. In CosBench, 99.27% of answers contain API names mentioned in the queries.

The above empirical findings drove us to explore deeper relations between the types of queries and intentions and the effectiveness of code search methods.

1) Results w.r.t. *query intentions:* As Table IX shows, for queries on *reusing code*, Code-nn achieves a *Precision*@20 value of 0.3612, while among the four IR-based methods, CodeHow achieves the highest *Precision*@20 value (0.2898). Code-nn achieves higher *MAP*@20 and *MRR*@20 values than the IR-based methods.

There can be two reasons for this.

First, DL-based methods are much more suitable for searching semantically similar code snippets. For queries on *reusing code*, the words used in a query and those used in the answers tend to be semantically similar, rather than syntactically similar. For instance, let "query1: `How should I compare two strings?`" and "query2: `How can I check the equality of two strings?`" be the queries to be performed. Let an answer be the following:

```
void stringEquals(String a, String b){
  if (a == null) return (b == null);
  else return (a.equals(b));
}
```

The answer is semantically relevant to both of the queries, but the syntactical similarity between `stringEquals` and query1 is lower than that between `stringEquals` and query2 due to the words used. Code-nn improves code search in that it learns more semantic relations among words using a deep learning model.

Second, Code-nn takes function names as code features. Function names tend to reflect functional behaviors of code snippets. Thus Code-nn outperforms the others, as queries on *reusing code* often contain function names.

For queries on *resolving bugs* and *using APIs*, Lucene achieves *Precision*@20 values that are 4.23× and 2.2× higher than those of Code-nn, respectively. Similarly, the IR-based methods achieve higher *MAP*@20 and *MRR*@20 values than the DL-based methods. One main reason is that these queries do contain code information, such as API names, exception names, and buggy code, which facilitates the IR-based methods in matching code with the queries.

The above results are summarized as the next observation.

Observation 6. *For queries on reusing code, Code-nn outperforms the other methods. For queries on using API and resolving bugs, IR-based methods outperform DL-based methods.*

2) Results w.r.t. *queries of different representations (keyword versus phrase):* As Table IX shows, the *Precision*@20, *MAP*@20, and *MRR*@20 values *w.r.t.* keyword queries are $1.1 \sim 3.1\times$ higher than those *w.r.t.* phrase queries, indicating that all of the code search methods are not effective in identifying and leveraging key information in phrase queries.

It is intuitively that a code search method for keyword queries may be able to match code with queries more easily to retrieve results, but a programmer may need to hold more domain knowledge when he/she raises a keyword query. Phrase queries do provide programmers more flexibility in asking unclear queries, while it can be more challenging for code search methods to obtain the real needs in phrase queries to retrieve more relevant search results.

TABLE IX
RESULTS *w.r.t.* DIFFERENT TYPES OF QUERY INTENTIONS AND REPRESENTATION.

Method	Code Reuse			Bug Resolution			API Learning			Phrase			Keyword		
	P@20	MAP@20	MRR@20	P@20	MAP@20	MRR@20	P@20	MAP@20	MRR@20	P@20	MAP@20	MRR@20	P@20	MAP@20	MRR@20
Lucene	0.2375	0.0825	0.3632	**0.3433**	0.1037	0.2712	**0.3250**	0.114	**0.3149**	0.2309	0.0662	**0.2753**	**0.3624**	0.1333	**0.3802**
LuSearch	0.0506	0.0192	0.1197	0.1011	0.0185	0.065	0.1233	0.0540	0.0615	0.0518	0.0145	0.0561	0.1231	0.0446	0.1256
CodeHow	0.2898	0.1123	0.2786	0.3255	**0.1326**	**0.2990**	0.3227	0.1179	0.2923	**0.2644**	0.0880	0.2415	0.3609	**0.1566**	0.3427
QECK	0.1749	0.0667	0.2125	0.2122	0.1150	0.2868	0.2273	**0.1232**	0.3073	0.1925	0.0608	0.1767	0.2124	0.0949	0.2624
YeSearch	0.103	0.0253	0.1714	0.2100	0.0475	0.1130	0.2247	0.0501	0.1732	0.1260	0.0263	0.1293	0.2141	0.0525	0.1851
Code-nn	**0.3612**	**0.1701**	**0.5030**	0.0811	0.0195	0.1774	0.1476	0.0679	0.1614	0.2032	**0.0928**	0.2598	0.2458	0.1065	0.3793

TABLE X
PROS AND CONS OF VARIOUS CODE SEARCH METHODS

	Pros	**Cons**	**Int.**	**Rep.**
Lucene	simple and stable.	cannot handle complex situations.	*API, Bug*	Key word
LuSearch	-	ineffective.	-	-
CodeHow	use API to expand query to reduce the differences between query and code.	remains a problem which libraries should be included.	*API, Bug*	Key word
QECK	employs QAs to expand queries to reduce the differences between query and code.	introduce irrelevant words into query expansions.	-	-
YeSearch	learns a word embedding for code.	ineffective.	-	-
Code-nn	leverage semantic information for code searches.	poor for query for API and Bug.	*Reuse*	Key word

Thus we draw the next observation.

Observation 7. *Most of the existing code search methods perform better on keyword queries than on phrase queries.*

D. Summary and Feedback

Table X summarizes pros and cons of the code search methods. IR-based methods are popular in industry because they are easier to implement, and their processes and intermediate results are more intuitive to understand. On the other hand, IR-based methods are usually less efficient for processing synonyms that occur frequently in queries. In contrast, DL-based methods can process more complicated queries, while they do need model training, and the training datasets also affect the effectiveness of code search. Further, the models are usually incomprehensible, and are not easy to evolve.

V. THREADS TO VALIDITY

Threats to internal validity are primarily related to uncontrolled internal factors that may affect the evaluation results. First, our implementations of the four code search methods may be inconsistent with those implemented by their authors; errors may also be latent in the implementations of code search methods. As the implementations of four selected methods are not publicly available, we re-implemented them by faithfully following the principle of each method, but took some strategies to simplify the implementations—we let Lucene be the rendering framework, and only implemented the specific strategies and algorithms taken by the other methods. Mature third-party libraries, such as Keras and Wordnet, were employed for preventing defects during the implementation phase.

Second, ground-truth answers were prepared for each query manually; bias may exist for determining whether a code snippet should be a ground-truth answer to a query. We reduced the bias by inviting five developers of different development experiences and domain knowledge to repeat the manual vetting process and verified the answers by their voting on the correctness of each answer.

Threats to external validity are mainly related with the generalizability of the observations. CosBench was specifically built and evaluated on the chosen code search methods; the observations may not be valid when other datasets or code search methods were employed. Nevertheless, this study has the following advantages that may render better generalizability than other previous studies in the literature: (1) The codebase was established by following a general process that has been popularly used in the MSR (Mining Software Repositories) community [9], [25], [31], [52]; (2) The natural-language queries were obtained from popular posts on Stack Overflow, which is fair and has also been adopted in many other researches [37], [38], [48]; (3) The four metrics have also been used in many researches, which have been explained in Section II; and (4) Six methods belonging to two mainstreams of code search methods that support natural-language queries were chosen for evaluations.

VI. RELATED WORK

A. Code Search

Researchers have proposed many IR-based and DL-based natural-language code search methods [8]. Wu *et al.* [47] have proposed a method to predict the alteration intent and use it for query expansion. Lu *et al.* [23] have proposed INQRES, which considered the relations between words in the source code to optimize the query quality. Rahman *et al.* [34] have proposed a technique that reformulates the query by relevant API from StackOverflow. Zhang *et al.* [51] have proposed an approach to find identifiers that are semantically related to a given natural-language query. Sirres *et al.* [41] have presented CoCaBu which resolves the vocabulary mismatch problem when dealing with free-form code search queries. Vinayakarao *et al.* [45] have proposed the ANNE approach to discover the mappings between syntactic forms and programming concepts and expanding codebases. Allamanis *et al.* [9] have learned a bimodal model conditioned on natural-language text, and used it to rank code search results. Niu *et al.* [32] have used learning-to-rank methods to automatically train a ranking schema.

Several code search studies focus on automatically generating and/or ranking code results based on partial search results and

TABLE XI
EXISTING DATASETS IN THE LITERATURE.

Source	Description	Language	Research Use	Note
Zhang *et al.* [52]	65,253 projects, 78,165,560 snippets	C, C++, C#, Java, JS	Code search	Not accessible
Gu *et al.* [13]	9,950 projects, 16,262,602 methods	Java	Code search	Not accessible
Ye *et al.* [49]	Four open source project	Java	Code search	A set of bug reports
Lv *et al.* [25]	26K C# projects, 8.3M C# code files and 11.4M methods	C#	Code search	Not accessible
Nie *et al.* [31]	1,538 projects, 921,713 code snippets	Android/Java	Code search	A set of Android specific code snippets
Li *et al.* [21]	24,549 GitHub repositories, 4,716,814 methods	Andriod/Java	Code search	A set of Android specific code snippets
Hamel *et al.* [15]	2 million (comment, code) pairs from open source libraries.	Python, JS, Ruby, Go, Java, and PHP	Code search	Basic, multiple code language dataset.
Yin *et al.* [50]	2,379 training and 500 test examples, 600k mining example	Java/python	General purposed	A dataset of low quality
Iyer *et al.* [16]	145,841 pairs of C# and 41,340 pairs of SQL	C#,SQL	General purposed	A small number of code snippets

a large amount of historical data. Moreno *et al.* [30] have proposed MUSE, a method for mining and sorting code examples. Raghothaman *et al.* [33] have proposed SWIM, which translated user queries into the APIs and synthesized idiomatic code describing the uses of these APIs. Galenson *et al.* [12] have proposed CodeHint for synthesizing dynamic code. Martie *et al.* [27] have developed CodeExchange, which explicitly leverages contexts to support fluid, expressive reformulation of queries. Martie *et al.* [26] have also introduced a search engine, CodeLikeThis, which can directly use the results of previous queries to conduct the succeeding queries. Wang *et al.* [46] have proposed a model allowing user demands to be enforced for filtering and/or optimizing code search results.

Code search engines based on IDE have been developed. Rahman *et al.* [35] have developed RACK, which automatically mines relevant code snippets from thousands of open source projects and displays them as sorted lists in IDEs. Zhang *et al.* [52] have developed Bing Developer Assistant (BDA) that improves developers' productivity by recommending sample code retrieved from software repositories and web pages. Campbell *et al.* [11] have introduced a content-assisted tool named NLP2Code that saves developers' efforts in switching from their IDEs to web browsers when they need to search.

B. Existing Datasets for Code Search

Many datasets of code snippets and natural-language queries do exist, while they are mainly used for data training and validation. Zhang *et al.* [52], Gu *et al.* [13], Ye *et al.* [49], Lv *et al.* [25], Nie *et al.* [31], Husain *et al.* [15], Li *et al.* [21] have created their own datasets in their studies. Others, such as Yin *et al.* [50], Iyer *et al.* [16], have also provided datasets with natural-language queries and code snippets answers, which are also promising for evaluating code search.

The above datasets, as Table XI shows, often contain: (1) code snippets and comments contributed to the GitHub platform, (2) QA pairs collected from the SO platform, where the answers may contain many code snippets, (3) online documents, such as tutorials, JDK documents, *etc.*, and (4) the others, such as bug reports, *etc.* Comparatively, CosBench is specifically built for measuring natural-language code search, as it does provide not only the codebase, but also a set of queries and the ground truths on which code search methods can be evaluated fairly. Furthermore, CosBench contains queries of different intentions, assisting engineers in evaluating and choosing search techniques for their respective intentions.

We did not evaluate all natural-language code search methods on CosBench because of our limited resources for implementing all those search methods. Nevertheless, the six search methods presented in this paper can be the representative baselines, and researchers can propose and evaluate their own search methods on CosBench. We believe that CosBench is flexible and extensive so that it can be equipped with new code search strategies, which also encourages researchers to propose more effective and efficient strategies and evaluate them on the CosBench dataset.

VII. CONCLUSION

This paper presents an empirical study of code search methods that use natural-language queries as input. We have created a CosBench dataset, which currently consists of $4,199,769$ code snippets from 1000 Java projects on GitHub, 52 queries with ground truths and of three different types of intentions, and four metrics, for evaluating code search methods. We have also implemented four representative code search methods and evaluated them against Lucene and Code-nn on the CosBench dataset. The empirical results clearly show the usefulness of the CosBench dataset and the strength of each code search method. In particular, we have observed that code search methods can be selectively applied to perform queries of different types of intentions, and Deep Learning-based methods are more suitable for queries on *reusing code*, and Information Retrieval-based ones for queries on *resolving bugs* and *learning API uses*. This observation also leads to a piece of interesting future work—the intention of a query from a programmer may be inferred automatically so as to help the programmer to use a suitable search method.

ACKNOWLEDGEMENT

This research was sponsored by the National Key Research and Development Program of China (Project No. 2018YFB1003903), National Nature Science Foundation of China (Grant No. 61472242 and 61572312), and the Singapore Ministry of Education under its Academic Research Fund Tier 2 Grant (MOE2019-T2-1-193). We would like to thank the anonymous reviewers for valuable feedback on earlier drafts of this paper. We would also like to thank the code search developers for analyzing and evaluating our dataset.

REFERENCES

[1] Bm25. https://en.wikipedia.org/wiki/Okapi_BM25. Accessed April, 2019.

[2] Github. https://github.com/search?q=. Accessed April, 2019.

[3] Google Scholar. https://scholar.google.com/. Accessed April, 2019.

[4] Krugle. http://www.krugle.com/. Accessed April, 2019.

[5] lucene. http://lucene.apache.org/. Accessed April, 2019.

[6] searchcode. https://searchcode.com/. Accessed April, 2019.

[7] StackOverflow. https://stackoverflow.com/. Accessed April, 2019.

[8] M. Allamanis, E. T. Barr, P. Devanbu, and C. Sutton. A survey of machine learning for big code and naturalness. *ACM Comput. Surv.*, 51(4):81:1–81:37, July 2018.

[9] M. Allamanis, D. Tarlow, A. Gordon, and Y. Wei. Bimodal modelling of source code and natural language. In *International Conference on Machine Learning*, pages 2123–2132, 2015.

[10] V. Balachandran. Query-by-example in large-scale code repositories, Apr. 19 2016. US Patent 9,317,260.

[11] B. A. Campbell and C. Treude. NLP2Code: Code snippet content assist via natural language tasks. In *ICSME*, pages 628–632. IEEE, 2017.

[12] J. D. Galenson. *Dynamic and Interactive Synthesis of Code Snippets*. PhD thesis, UC Berkeley, 2014.

[13] X. Gu, H. Zhang, and S. Kim. Deep code search. In *ICSE*, pages 933–944. IEEE, 2018.

[14] S. Hochreiter and J. Schmidhuber. Long short-term memory. *Neural computation*, 9(8):1735–1780, 1997.

[15] H. Husain, H.-H. Wu, T. Gazit, M. Allamanis, and M. Brockschmidt. Codesearchnet challenge: Evaluating the state of semantic code search. *ArXiv*, abs/1909.09436, 2019.

[16] S. Iyer, I. Konstas, A. Cheung, and L. Zettlemoyer. Summarizing source code using a neural attention model. In *Proceedings of the 54th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, volume 1, pages 2073–2083, 2016.

[17] B. J. Jansen and S. Y. Rieh. The seventeen theoretical constructs of information searching and information retrieval. *Journal of the American Society for Information Science and Technology*, 61(8):1517–1534, 2010.

[18] I. Keivanloo, J. Rilling, and Y. Zou. Spotting working code examples. In *ICSE*, pages 664–675. ACM, 2014.

[19] K. Kim, D. Kim, T. F. Bissyandé, E. Choi, L. Li, J. Klein, and Y. L. Traon. FaCoY: a code-to-code search engine. In *ICSE*, pages 946–957, 2018.

[20] J. R. Koza, F. H. Bennett, D. Andre, and M. A. Keane. Automated design of both the topology and sizing of analog electrical circuits using genetic programming. In *Artificial Intelligence in Design*, pages 151–170. Springer, 1996.

[21] H. Li, S. Kim, and S. Chandra. Neural code search evaluation dataset. *ArXiv*, abs/1908.09804, 2019. https://github.com/facebookresearch/Neural-Code-Search-Evaluation-Dataset.

[22] H. Li, Z. Xing, X. Peng, and W. Zhao. What help do developers seek, when and how? In *WCRE*, pages 142–151. IEEE, 2013.

[23] J. Lu, Y. Wei, X. Sun, B. Li, W. Wen, and C. Zhou. Interactive query reformulation for source-code search with word relations. *IEEE Access*, 6:75660–75668, 2018.

[24] M. Lu, X. Sun, S. Wang, D. Lo, and Y. Duan. Query expansion via wordnet for effective code search. In *SANER*, pages 545–549, 2015.

[25] F. Lv, H. Zhang, J.-g. Lou, S. Wang, D. Zhang, and J. Zhao. CodeHow: Effective code search based on api understanding and extended boolean model (e). In *ASE*, pages 260–270. IEEE, 2015.

[26] L. Martie, A. v. d. Hoek, and T. Kwak. Understanding the impact of support for iteration on code search. In *11th Joint Meeting on Foundations of Software Engineering*, pages 774–785. ACM, 2017.

[27] L. Martie, T. D. LaToza, and A. van der Hoek. CodeExchange: Supporting reformulation of internet-scale code queries in context (t). In *ASE*, pages 24–35, 2015.

[28] R. Mihalcea, C. Corley, C. Strapparava, et al. Corpus-based and knowledge-based measures of text semantic similarity. In *AAAI*, volume 6, pages 775–780, 2006.

[29] T. Mikolov, K. Chen, G. Corrado, and J. Dean. Efficient estimation of word representations in vector space. *arXiv preprint arXiv:1301.3781*, 2013.

[30] L. Moreno, G. Bavota, M. Di Penta, R. Oliveto, and A. Marcus. How can I use this method? In *ICSE*, pages 880–890, 2015.

[31] L. Nie, H. Jiang, Z. Ren, Z. Sun, and X. Li. Query expansion based on crowd knowledge for code search. *IEEE Transactions on Services Computing*, 9(5):771–783, 2016.

[32] H. Niu, I. Keivanloo, and Y. Zou. Learning to rank code examples for code search engines. *Empirical Software Engineering*, 22(1):259–291, 2017.

[33] M. Raghothaman, Y. Wei, and Y. Hamadi. Swim: Synthesizing what I mean-code search and idiomatic snippet synthesis. In *ICSE*, pages 357–367, 2016.

[34] M. M. Rahman and C. Roy. Effective reformulation of query for code search using crowdsourced knowledge and extra-large data analytics. In *ICSME*, pages 473–484, 2018.

[35] M. M. Rahman, C. K. Roy, and D. Lo. RACK: Code search in the IDE using crowdsourced knowledge. In *ICSE Companion*, pages 51–54, 2017.

[36] F. Rosenblatt. Principles of neurodynamics. perceptrons and the theory of brain mechanisms. Technical report, Cornell Aeronautical Lab Inc Buffalo NY, 1961.

[37] C. Sadowski, K. T. Stolee, and S. Elbaum. How developers search for code: a case study. In *ESEC/FSE*, pages 191–201, 2015.

[38] S. E. Sim, M. Agarwala, and M. Umarji. A controlled experiment on the process used by developers during internet-scale code search. In *Finding Source Code on the Web for Remix and Reuse*, pages 53–77. Springer, 2013.

[39] S. E. Sim and R. E. Gallardo-Valencia. *Finding source code on the web for remix and reuse*. Springer, 2013.

[40] S. E. Sim, M. Umarji, S. Ratanotayanon, and C. V. Lopes. How well do search engines support code retrieval on the web? *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 21(1):4, 2011.

[41] R. Sirres, T. F. Bissyandé, D. Kim, D. Lo, J. Klein, K. Kim, and Y. Le Traon. Augmenting and structuring user queries to support efficient free-form code search. *Empirical Software Engineering*, 23(5):2622–2654, 2018.

[42] K. Sparck Jones. A statistical interpretation of term specificity and its application in retrieval. *Journal of documentation*, 28(1):11–21, 1972.

[43] K. T. Stolee, S. Elbaum, and D. Dobos. Solving the search for source code. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 23(3):26, 2014.

[44] I. Sutskever, O. Vinyals, and Q. V. Le. Sequence to sequence learning with neural networks. In *Advances in neural information processing systems*, pages 3104–3112, 2014.

[45] V. Vinayakarao, A. Sarma, R. Purandare, S. Jain, and S. Jain. Anne: Improving source code search using entity retrieval approach. In *Proceedings of the Tenth ACM International Conference on Web Search and Data Mining*, pages 211–220. ACM, 2017.

[46] S. Wang, D. Lo, and L. Jiang. Active code search: incorporating user feedback to improve code search relevance. In *ASE*, pages 677–682, 2014.

[47] H. Wu and Y. Yang. Code search based on alteration intent. *IEEE Access*, 7:56796–56802, 2019.

[48] X. Xia, L. Bao, D. Lo, P. S. Kochhar, A. E. Hassan, and Z. Xing. What do developers search for on the web? *Empirical Software Engineering*, 22(6):3149–3185, 2017.

[49] X. Ye, H. Shen, X. Ma, R. Bunescu, and C. Liu. From word embeddings to document similarities for improved information retrieval in software engineering. In *ICSE*, pages 404–415, 2016.

[50] P. Yin, B. Deng, E. Chen, B. Vasilescu, and G. Neubig. Learning to mine aligned code and natural language pairs from stack overflow. In *MSR*, pages 476–486, 2018.

[51] F. Zhang, H. Niu, I. Keivanloo, and Y. Zou. Expanding queries for code search using semantically related api class-names. *IEEE Transactions on Software Engineering*, 44(11):1070–1082, 2018.

[52] H. Zhang, A. Jain, G. Khandelwal, C. Kaushik, S. Ge, and W. Hu. Bing developer assistant: improving developer productivity by recommending sample code. In *FSE*, pages 956–961. ACM, 2016.

[53] S. Zhou, B. Shen, and H. Zhong. Lancer: Your code tell me what you need. In *ASE Demonstrations*, 2019.