Deep-Learning-Based Code Search: The Road Ahead

Chao Liu, Xin Xia, David Lo, Zhiwei Liu, Ahmed E. Hassan, and Shanping Li

Abstract—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, which match keywords in query with code text. But they fail to connect the semantic gap between query and code. To conquer this challenge, Gu et al. proposed a deep-learning-based model named DeepCS. It jointly embeds method code and natural language description into a shared vector space, where methods related to a natural language query are retrieved according to their vector similarities. Although their reported experimental results are promising, there remain questions about why it works well. Thus, this study performed an in-depth analysis and found that: (1) The performance of DeepCS is unstable for code search. (2) DeepCS works poorly on a codebase with new/updated repositories due to its overfitting issue; although re-training DeepCS is a solution, its computation complexity is too high for practical usage.

To overcome DeepCS' issues, we proposed a simplified model CodeMatcher that leverages the IR technique but maintain many features in DeepCS. Generally, CodeMatcher combines query keywords with the original order, performs a fuzzy search on name and body strings of methods, and returned the best-matched methods with the longer sequence of used keywords. We verified its effectiveness on a large-scale codebase with ˜41k repositories. Experimental results showed CodeMatcher outperforms DeepCS by 97% in terms of MRR (a widely used accuracy measure for code search) and it is over 66 times faster than DeepCS. We also provide some in-depth discussions and suggestions about the road ahead for code search.

Index Terms—code search, code indexing, program analysis.

1 Introduction

CODE search is the most frequent activity in software development [1], [2], [3] as developers favor searching existing code and learning from them just-in-time when meeting a programming issue [4], [5]. Reusing existing diverse technologies and complex frameworks from millions of open-source repositories (e.g., in GitHub[1]) can maximize developers' productivity [6], [7], [8], [9], [10]. During software development, it was observed that more than 90% of developers' code search efforts are used to find code snippet [11], thus this study focuses on searching code methods following previous studies [12], [13], [14] instead of searching repositories [15], [16].

Existing Challenges. Prior work on code search starts with leveraging information retrieval (IR) techniques (e.g., Koders[2] and Krugle[3]), which regards method code as text and match keywords in query with indexed methods [14], [17]. But their performance is poor due to two reasons: *(1) short and diverse queries,* keywords matching can hardly represent the search requirement due to insufficient context; *(2) representation of method as text,* method has structure that carries specialized semantics [11]. To address these issues, many past studies focus on query expansion and reformulation [9], [18], [19], [20], [21]. For example, the Sourcerer model [14] extended the textual content of a method with structural information; the model proposed by Lu et al. [20] expanded a query with synonyms generated from WordNet and matched keywords to method signatures; the CodeHow model [13] extended query with related APIs, and searched code methods with matched APIs and query keywords through an extended Boolean model.

Recently, Gu et al. [12] observed that IR-based models have two problems: *(1) semantic gap*, keywords cannot adequately represent high-level intent implied in the query, 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 gaps between query and code, Gu et al. [12] proposed a deep-learning-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 [14] and CodeHow [13].

However, we observed that more investigation needs to be made on DeepCS especially considering two aspects:

- *Chao Liu and Shanping Li are with the College of Computer Science and Technology, Zhejiang University, Hangzhou, China.*
  E-mail: {liuchaoo,shan}@zju.edu.cn
- *Xin Xia is with Faculty of Information Technology, Monash University, Melbourne, Australia.*
  E-mail: xin.xia@monash.edu
- *David Lo is with the School of Information Systems, Singapore Management University, Singapore.*
  E-mail: davidlo@smu.edu.sg
- *Zhiwe Liu is with Faculty of Baidu Inc., Shanghai, China.*
  E-mail: zhiweiliu03@baidu.com
- *Ahmed E. Hassan is with School of Computing, Queen's University, Ontario, Canada.*
  E-mail: ahmed@queensu.ca
- *Xin Xia is the corresponding author.*

Manuscript received April 00, 0000; revised August 00, 0000.

1. https://github.com/
2. www.koders.com
3. www.krugle.com

(1) uncontrollable black-box model, although Gu et al. [12] provided three concrete query examples to explain how it works, it is not enough to understand why it works so well; *(2) high computation complexity*, DeepCS works slowly, as it requires more than 50 hours for model training on a machine with an Nvidia K40 GPU, and developers need to wait for about 14s on a code search. Thus, it is necessary to explore the possibility to simplify DeepCS in a controllable way for a practical appliance.

Research Questions and Contributions. Based on the above considerations on DeepCS, this study aims to answer the following two research questions (RQs):

RQ1: Why does DeepCS perform so well? We investigated two important factors on the deep-learning-based model as follows:

(1) Model randomness. In the model training, the parameters in DeepCS were initialized by a pseudo-random generator and optimized by randomly selected training data for each optimization epoch. To explore how these two random factors affect model performance, we re-ran DeepCS 20 times by using the original source code[4] and dataset[5] released by the authors (with ˜10k repositories for testing code search). Results showed DeepCS is unstable since the MRR values of DeepCS range from 0.47 to 0.59 with mean 0.53, and its standard deviation 0.04 means a significant 8% of performance fluctuation. These results imply that applying DeepCS for real-world code search would not necessarily obtain the same high-level performance as reported by Gu et al. [12] with MRR=0.6.

(2) Model overfitting. Gu et al. [12] indicated that their training and testing data have some overlap and claimed that this threat to overfitting is negligible. To verify this assumption, we tested the 20 trained DeepCS on a larger-scale testing data with ˜41k repositories with no overlap with the original training data. Results showed that DeepCS has overfitting issues due to the out-of-vocabulary words in the new codebase that negatively impacts the performance of DeepCS. We observe that the best MRR is reduced by 39% from 0.59 to 0.36. Hence, the performance of DeepCS degrades as it is applied to search for relevant code methods from a larger codebase. Thus, DeepCS may not work well for dynamic codebases that grow over time like GitHub.

RQ2: Can a simpler and faster model outperform DeepCS? From the experimental results, we observed that DeepCS succeeded because it can correctly match query tokens with code method string. Inspired by this observation, we proposed a simplified IR-based model named CodeMatcher. Generally, it combines query keywords with the original order, performs a fuzzy search on name and body strings of methods, and returned the best-matched methods with longer sequence of used keywords. Experimental results on the larger-scale codebase showed that CodeMatcher substantially outperforms DeepCS, where its MRR is higher than the best MRR of DeepCS by 97% – i.e., 0.71 vs. 0.36. Besides, CodeMatcher needs no model training, and it only takes 0.21s for code search per query, which is 66 times faster than DeepCS.

Finally, we conducted a further analysis of the code search. We pointed out that matching query tokens with the method name and body strings is not enough for this task, especially for the queries with complex syntax structures. For the real-world code search, we just move one step further. For the road ahead, we suggested to combine the deep-learning-based model and the IR-based model and balance their pros and cons.

The main contributions of this work are as follows:

- We perform an in-depth analysis of Gu et al.'s DeepCS model on a larger-scale codebase collected by us;
- We propose a simple model CodeMatcher that outperforms DeepCS by more than 97% in terms of MRR and is 66 times faster than DeepCS.

Paper Organization. The remainder of this paper is organized as follows. Section 2 describes the background of the DeepCS model. Section 3 presents the experiment setup. Section 4 and 5 respectively show the answers to the research questions raised in Section 3, followed by the discussion in Section 6 and the suggestions for the road ahead in Section 7. Section 8 describes related work and Section 9 summarizes this study and presents future work.

2 Background

In the section, we present a brief background of the state-of-the-art model DeepCS proposed by Gu et al. [12]

Queries and Codebase. To simulate a real-world code search scenario, Gu et al. [12] manually collected 50 queries from Stack Overflow, as shown in Table 1. These queries are top-50 voted Java programming questions[6] following three criteria: *(1) concrete*, a question should be a specific Java programming task, such as "How can I concatenate two arrays in Java?"; *(2) well-answered*, the accepted answers corresponding to the question should contain a Java code snippet; *(3) non-duplicated*, the question is not a duplicate of another question in the same collection.

Moreover, Gu et al [12] extracted all methods from ˜10k GitHub repositories as codebase for testing code search. These repositories are Java projects with at least 20 stars. To perform code search, each method is represented by a <*method name, API sequence, token*> tuple. In the tuple, *method name* is a sequence of camel[7] split tokens derived from the name of the method, e.g., we split 'getFile' into 'get file'; Also, API sequence is a sequence of constructor invocation and methods calls in the method body; Moreover, *token* is a set of camel split tokens in method body, where duplicated tokens, stop words (e.g., 'the' and 'in') and Java keywords are removed.

DeepCS and Training Data. To obtain the semantic relationship between natural language in query and programming language in code, Gu et al.'s DeepCS [12] leverages deep learning to jointly embed the two languages in three steps:

(1) Tokenization, a natural language is represented by a sequence of English tokens (A). Meanwhile, a code is represented by three parts as referred above: camel split

4. https://github.com/guxd/deep-code-search
5. https://pan.baidu.com/s/1U_MtFXqq0C-Qh8WUFAWGvg
6. https://stackoverflow.com/questions/tagged/java?sort=votes
7. https://en.wikipedia.org/wiki/camelcase

TABLE 1: Benchmark queries collected by Gu et al. [12] for code search and FRank (i.e., the rank of the first correct result) values, where DCSr and DCSe are the DeepCS models tested on original data and our collected codebase respectively, with the best performance among 20 times of repetitions; CM is the proposed model CodeMatcher; GH is the GitHub Searcher[8]

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

token sequence of method name (B), a sequence of APIs (C), and a set of tokens (D).

(2) Joint embedding, To embed a method code as a characteristic vector $\boldsymbol{c}$, DeepCS uses two recurrent neural network (RNN) to embed tokens in method name and API sequence in method body respectively, and it leverages a multilayer perceptron (MLP) to embed tokens in method body with bag-of-words assumption. As to the natural language description $\boldsymbol{d}$, DeepCS utilizes a RNN to embed its tokens. In this way, the similarity between method and description can be measured by $cos(\boldsymbol{c}, \boldsymbol{d})$=$(\boldsymbol{c}^T \boldsymbol{d})/(\|\boldsymbol{c}\| \, \|\boldsymbol{d}\|)$.

(3) Optimization, the parameters in DeepCS is initialized by pseudo-random generators and optimized by randomly selected methods with related Javadoc comments. Specifically, the commented methods were extracted from GitHub Java repositories created from Aug. 2008 to Jun. 2016 with at least one star. During the training time, a training instance is a triple $<C$,D^+,$D^->$: for a code ($\boldsymbol{c}$$\in$$C$) and the first line in related Javadoc description ($\boldsymbol{d}^+$$\in$$D^+$), the model randomly chooses an incorrect description ($\boldsymbol{d}^-$$\in$$D^-$) from the pool of all D^+'s. Then, to optimize parameters θ in all neural networks, DeepCS predicts the cosine similarities of both $<C$,$D^+>$ and $<C$,$D^->$ pairs and minimize the loss function, $L(\theta)$=$\sum_{<C,D^+,D^->\in P} max(0, 0.05 - cos(\boldsymbol{c}, \boldsymbol{d}^+) + cos(\boldsymbol{c}, \boldsymbol{d}^-))$. Intuitively, the loss function encourages the correct description to a method $cos(\boldsymbol{c}, \boldsymbol{d}^+)$ and discourage the incorrect one $cos(\boldsymbol{c}, \boldsymbol{d}^-)$. Using the optimized model, the top-10 methods mostly related to a query are returned, in terms of cosine similarities between returned methods and query calculated by DeepCS.

3 Experiment Setup

3.1 Research Questions

Based on the state-of-the-art model DeepCS [12] for code search, this study investigated the following two research questions (RQs):

RQ1. Why does DeepCS perform so well?

Gu et al. [12] attributed DeepCS' success to the deep learning technique, but they left two critical questions on it: *(1) randomness,* a deep-learning-based model is highly affected by its random initialization and randomly selected training data [22], [23], thus it is necessary to verify whether re-running DeepCS many times can obtain the same level of performance reported by authors. *(2) overfitting,* there are some overlapping between DeepCS' training and testing data. Although Gu et al. [12] claimed the threat for overfitting is not significant, it is worth to explore how DeepCS works on a larger-scale codebase without overlapping with training data.

RQ2. Can a simpler and faster model outperform DeepCS?

The working process of DeepCS [12] is complicated and time-consuming, which requires more than 50 hours of training time running on a server with one Nvidia K40 GPU, and developers need to wait for about 14s on a code search.

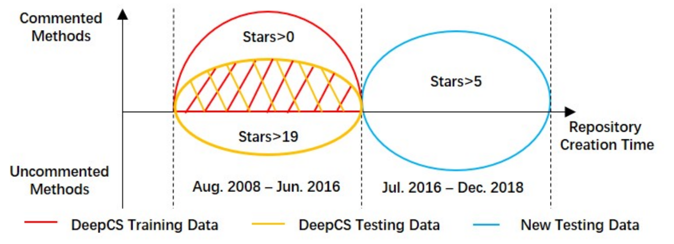


Fig. 1: Illustration of DeepCS' training and testing data, and the newly collected testing data.

TABLE 2: Statistics of constructed codebase.

#Project	#Method	#Javadoc
41,260	16,611,025	3,639,794

Simple and fast model is easier to be adopted in practice, especially for the GitHub whose projects are growing and changing every second. Therefore, this study investigated the possibility to simplify DeepCS by using an IR-based technique.

3.2 Dataset

Codebase. To construct a larger-scale codebase without overlapping with DeepCS' training data, we crawled 41,260 Java repositories from GitHub. These projects were created from Jul. 2016 to Dec. 2018 with more than five stars. As shown in Fig. 1, the overlapped data in DeepCS is the commented methods in repositories created during Aug. 2008 and Jun. 2016 with stars>19, and the time duration of our new codebase can ensure the non-overlapping with DeepCS' training data. From Table 2, we can observe that the new codebase contains ~17 million methods, and 21.91% of them have Javadoc comments that describe the corresponding methods.

Queries. Following Gu et al. [12], we validated a code search model with 50 queries as model inputs as referred to in Section 2. And the used queries are listed in Table 1.

3.3 Evaluation Criteria

To measure the effectiveness of code search, we utilized four common evaluation metrics following prior code search studies [12], [13], [24], [25], [26], including FRank, Recall@k, Precision@k, and Mean Reciprocal Rank (MRR).

FRank, the rank of the first correct result in the result list [25]. It measures users' inspection effort for finding the expected method code when scanning the candidate list from top to bottom. A smaller FRank value implies users' lower efforts and better effectiveness for a single code search query.

Recall@k, the percentage of queries for which more than one correct results exist in the top-k ranked results [24], [26], [27]. Specifically, $Recall@k = Q^{-1}\sum_{q=1}^{Q}\delta(FRank_q \leq k)$, 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 Recall@k means better code search performance, and users can find desired method by inspecting fewer returned code list.

Precision@k, the average percentage of relevant results in top-k returned code 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 [12]. Precision@k is useful and important because users often hit multiple responded results for learning different code usages [25]. Larger Precision@k indicates that less noisy results are returned by a code search model.

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 hit result ($FRank$) [12], [24], [26]. Thus, the formula is $MRR = Q^{-1}\sum_{q=1}^{Q} FRank_q^{-1}$. MRR is a comprehensive evaluation for the code search, and larger MRR value means better performance.

Note that, following Gu et al. [12], FRank and MRR are calculated within the top-10 results, as users commonly desired to find results from the top-10 list. Meanwhile, the cutoff coefficients k in $Recall@k$ and $Precision@k$ are set to 1, 5, and 10 respectively, which reflect the typical sizes of results that users inspect [24].

4 RQ1: Why Does DeepCS Perform So Well?

4.1 Analyzing Model Randomness

To analyze the how the model randomness affects its performance, we re-ran DeepCS 20 times by using the source code shared in GitHub[9] and the complete dataset[10] provided by the authors [12]. Following the default setting, we trained DeepCS with 500 epochs and performed code search on their codebase.

Fig. 2(a) shows that the MRR values of re-ran model (DeepCSr) range from 0.47 to 0.59 with mean 0.53, and its standard deviation 0.04 indicates a significant 8% of performance fluctuation. Thus, the DeepCS model is unstable due to its randomly initialized parameters and randomly selected training data for each optimization epoch.

Table 3 shows that the best performance of DeepCSr (MRR=0.59) among 20 repetitions have Recall@1/5/10 = 0.44/0.72/0.82 and Precision@1/5/10 = 0.44/0.42/0.41. In comparison, the original results reported by Gu et al. [12] (DeepCSo) shows a similar performance with MRR=0.6, Recall@1/5/10 = 0.46/0.76/0.86, and Precision@1/5/10 = 0.46/0.5/0.49. However, the MRR of DeepCSo is far away from the mean value 0.53, with 13% difference. This phenomenon implies that applying DeepCS for real-world code search would not necessarily obtain the same high-level performance as reported by Gu et al. [12].

Moreover, we may notice that this randomness issue is not the only case for the deep-learning-based model but a widely accepted tough problem [22], [23]. Thus, to make a fair evaluation, it is suggested to supply repeated results with mean and variance for deep-learning-based models.

9. https://github.com/guxd/deep-code-search
10. https://pan.baidu.com/s/1U_MtFXqq0C-Qh8WUFAWGvg

TABLE 3: Overall accuracy (R, Recall; P, Precision; MRR) of CodeMatcher, GitHub Searcher, and DeepCS models with different settings.

Model	Description	R@1	R@5	R@10	P@1	P@5	P@10	MRR
DeepCSo	The original results of DeepCS reported by Gu et al. [12]	0.46	0.76	0.86	0.46	0.50	0.49	0.60
DeepCSr$_{best}$	The best results of DeepCS model re-ran by us among 20 repetitions.	0.44	0.72	0.82	0.44	0.42	0.41	0.59
DeepCSe$_{best}$	The best results of DeepCS model tested on larger-scale codebase but excluded the out-of-vocabulary tokens.	0.22	0.46	0.64	0.22	0.23	0.22	0.36
CodeMatcher	The proposed model in this study.	0.64	0.76	0.76	0.64	0.58	0.57	0.71
CodeMatcher-S_{body}	The proposed model in this study without using the S_{body} for code search.	0.60	0.72	0.74	0.60	0.53	0.50	0.68
GitHub Searcher	The code searcher in GitHub.	0.28	0.60	0.64	0.28	0.21	0.17	0.44

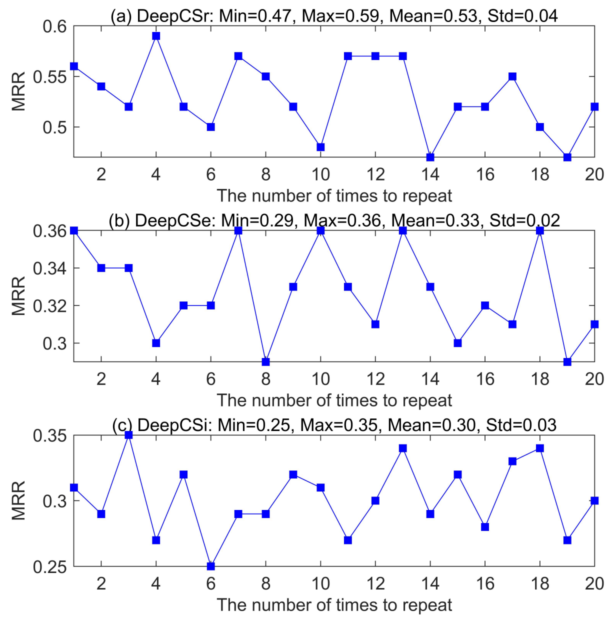


Fig. 2: Statistics of three DeepCS models running 20 times, where DeepCSr is the re-ran Gu et al's model [12]; DeepCSe is a DeepCS model running on new testing data excluding new tokens.

4.2 Analyzing Model Overfitting

To analyze the generalizability of DeepCS, we tested above 20 trained DeepCS on a larger-scale codebase without overlapping on its training data, as referred to in Section 3.2. According to DeepCS' requirement [12], Each method in the new codebase was parsed into a <*method name, API sequence, token*> tuple. Specifically, we leveraged the Javaparser[11] tool to transform a method into an abstract syntactic tree (AST), then traversed the AST to extract these three method components, and finally utilized python natural language toolkit (NLTK)[12] to address stemming, removing stop words, and performing camel split. Details described in Section 3.2.

Before searching code, DeepCS performs encoding on tokens of parsed code. To mitigate the model complexity, Gu et al. [12] limited the size of the vocabulary to 10,000 words that are frequently used in their training data. Table 4 shows that limited vocabulary can cover 98.51% tokens and 88.51% methods in the new codebase. However, due to the large size of the codebase, the vocabulary still missed ˜4.3 millions of new tokens on ˜1.9 millions of methods. Thus, it is a problem how to cope with the new tokens, excluding or including them? We respectively analyze these two situations as follows:

11. https://github.com/javaparser/javaparser
12. http://www.nltk.org/

TABLE 4: Statistics of new tokens and methods in the newly collected larger-scale codebase.

Type	#Total	#New	%New
Token	286,521,621	4,261,517	01.49%
Method	16,611,025	1,909,227	11.49%

(1) Excluding new tokens. It is natural to exclude the out-of-vocabulary tokens (DeepCSe) as they are noises for the model. Fig. 2(b) shows that the MRR values of DeepCSe range from 0.29 to 0.36, with mean 0.33 and standard deviation 0.02 (6% of performance fluctuation). Comparing with the DeepCSr, the mean MRR reduced by 38% from 0.53 to 0.33 while the best MRR decreased by 39% from 0.59 to 0.36.

(2) Including new tokens. Also, We investigated the noise tolerance of DeepCS by including new tokens in th codebase to the vocabulary. The results of this setting (DeepCSi) in Fig. 2(c) shows that its MRR ranges from 0.25 to 0.35, with mean 0.3 and standard deviation 0.03 (10% of performance fluctuation). Comparing with DeepCSe, the best MRR value of DeepCSi is similar (i.e., 0.36 vs. 0.35). But the mean MRR value slid by 9% from 0.33 to 0.3.

Above results indicate that DeepCS shows substantially lower performance on the new codebase without overlapping on its training data, implying overfitting of DeepCS in Gu et al.'s [12] experiment setup. Besides, DeepCS fails to work on the larger-scale codebase because it cannot handle out-of-vocabulary words. Therefore, DeepCS may not work well for dynamic codebases that grow over time like GitHub.

Result 1: *DeepCS performs so well not only because of its better representation and understanding between query and code through deep learning, but also for its unstable training process, model overfitting, and static codebase.*

5 RQ2: Can a Simpler and Faster Model Outperform DeepCS?

5.1 CodeMatcher

Generally, the proposed IR-based model CodeMatcher is a simplification of DeepCS, which takes two phases for code search:

Phase-I: Indexing Code. DeepCS represents a code as a <*method name, API sequence, token*> tuple, as referred to in Section 2. Similarly, we represented code as a <*method name, token sequence*> tuple: *(1) Method name.* We extract the string of method name without any processing, e.g., stemming. *(2) Token sequence.* A sequence of fully qualified tokens [13] (e.g., Java.lang.String or System.io.File.readlines()) in method parameters, method body, and returned parameter, where the token sequence can be regarded as a combination of <API sequence, token> in DeepCS. To parse a method, we transformed the method into an abstract syntactic tree (AST) with the Javaparser[13] library. And it is easy to extract programming items and infer their fully qualified tokens by traversing the AST like DeepCS [12]. Moreover, To build a method-level code search engine, we leveraged the Elasticsearch[14], a Lucene-based text search engine, to index method tuples and corresponding source code.

Phase-II: Matching Method with Keywords. DeepCS leveraged the joint embedding to learn the relationship between query and method represented by <*method name, API sequence, token*> tuple. In the view of IR-based model, we assume that DeepCS successively matches keywords in query with the method name and method body successively, and the joint embedding aims to generate correct keywords and replace keywords with synonyms at proper time. To simulate this theory, the proposed model CodeMatcher performs code search in three steps as follows:

Step-1: Token metadata. To decide the importance of query tokens for keywords, we generate metadata for each token. A token's metadata is represented by <*property, frequency, importance*> tuples: *(1) property*, the class of word (e.g., verb) identified by the Stanford Parser[15]; *(2) frequency*, the occurrence number of the camel[16] split tokens in method names in the codebase; *(3) importance*, the worth for method naming, which is divided by five levels as Table 5 based on our programming experience on naming conventions.

With the token metadata, we extract the core semantics in a query by removing the tokens not commonly used for method naming, and use the rest of tokens as candidate keywords. In specific, 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 focuses on Java projects. Moreover, We removed the level-1 tokens because they are seldom used for method naming, and stemmed the rest of tokens. If the frequency of a token is zero, we used its synonyms that generated by the WordNet[17] with the highest frequency as the substitution.

TABLE 5: Five token importance levels for method naming based on the token property (e.g., verb or noun) and whether the token is a class name in JDK.

Level	Condition
5	JDK Noun
4	Verb or Non-JDK Noun
3	Adjective or Adverb
2	Preposition or Conjunction
1	Other

Step-2: Keywords for code search. Using the filtered query tokens in Step-1, we launched a fuzzy match on indexed method names with a regular string[18]. The string is formed by all remaining query tokens in order as "$.*token_1.*\cdots.*token_n.*$". If the total number of returned methods is no more than 10, we removed the least important token with lower frequency one at a time, and performed the search again until only two tokens left. For each search round, we filtered out the repeated method by comparing the MD5 hash[19] values of their source code.

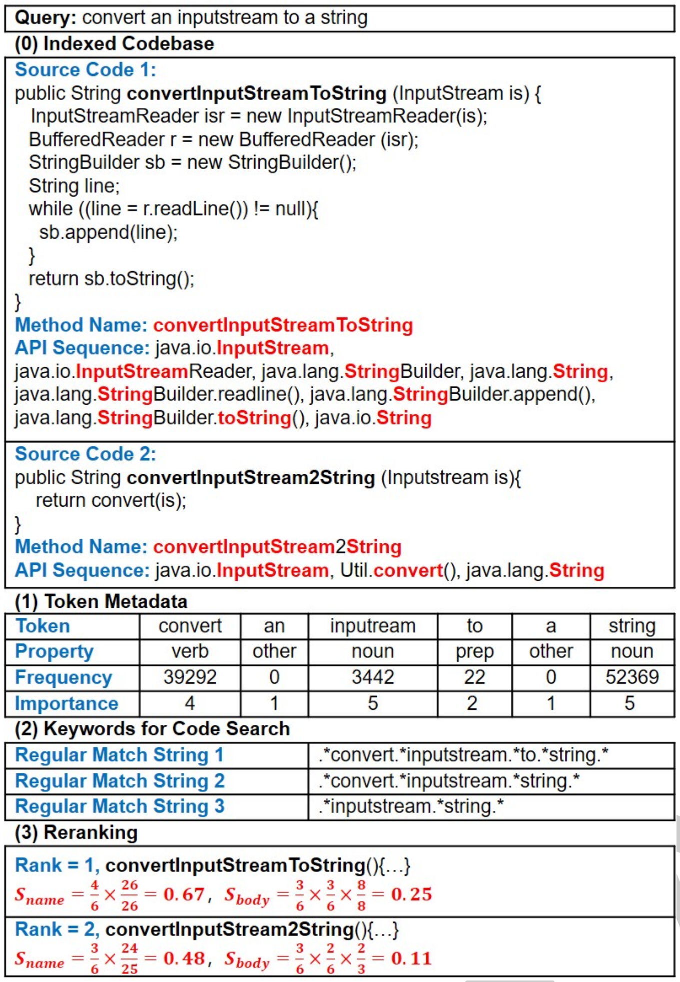
Step-3: Reranking. For optimization, we reranked the returned code list in descending order based on their matching scores (S_{name}), which measure the degree of token overlapping between query and method name as Eq. (1). Besides, if equal S_{name} exists, we boosted the rank of a method with a higher matching score (S_{body}) on between query and method body, as calculated in Eq. (2). Different from S_{name}, the last term in S_{body} is added to represent the ratio of JDK APIs in method, in terms of the fully qualified tokens, which assumes that developers favor a method with more JDK APIs. Finally, we returned the top-10 methods in the list.

$$S_{name} = \frac{\#query\ tokens\ as\ keywords}{\#query\ tokens} \times \frac{\#characters\ in\ name\ orderly\ matched\ keywords}{\#characters\ in\ name} \quad (1)$$

$$S_{body} = \frac{\#API\ tokens\ matched\ query\ tokens}{\#query\ tokens} \times \frac{Max[\#API\ tokens\ orderly\ matched\ query\ tokens]}{\#query\ tokens} \times \frac{\#JDKAPIs}{\#APIs} \quad (2)$$

Example. Fig. 3 illustrated an example for the first query *"convert an inputstream to a string"* in Table 1. 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>3442). With this metadata, CodeMatcher successively generates three 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}).

13. https://github.com/javaparser/javaparser
14. https://www.elastic.co/cn/downloads/elasticsearch
15. https://nlp.stanford.edu/software/lex-parser.shtml
16. https://en.wikipedia.org/wiki/camelcase
17. http://www.nltk.org/howto/wordnet.html
18. https://docs.python.org/3/library/re.html
19. https://docs.python.org/2/library/hashlib.html


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	inputream	to	a	string
Property	verb	other	noun	prep	other	noun
Frequency	39292	0	3442	22	0	52369
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.*

(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(){...}

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

Fig. 3: An example for CodeMatcher.

5.2 Evaluation

Searching Accuracy. Table 3 shows the experimental results of CodeMatcher on the larger-scale codebase. We can notice that its MRR=0.71 with Recall@1/5/10 = 0.64/0.76/0.76 and Precision@1/5/10 = 0.64/0.58/0.57. These results indicate substantial improvements over the best DeepCS model running on the same codebase (DeepCSe$_{best}$), where MRR increased by 97.22%, Recall@1/5/10 enhanced by 190.91%/65.22%/18.75% separately while Precision@1/5/10 boosted by 190.91%/152.17%/159.09% respectively. Moreover, even if we compare CodeMatcher with DeepCSr$_{best}$, the best DeepCS re-ran on the smaller-scale codebase, the proposed model also shows a substantial improvement where MRR improved by 20.34% from 0.59 to 0.71. These results proved the hypothesized theory on CodeMatcher raised in Section 5.1, and it is a good simplification of DeepCS.

Time Efficiency. Table 6 compares the time efficiency between CodeMatcher and DeepCSe. The DeepCSe is the re-ran DeepCS model trained by Gu et al.'s data [12] and tested on the same larger-scale codebase as CodeMatcher. Both models ran on a server with an Intel-i7 CPU and an Nvidia Geforce GTX1060 GPU. The results show that DeepCSe took 58.15h to train, 24.51h to preprocess codebase (i.e., parse, encode, and vectorize method name/APIs/tokens), and 695.03s to search code for 50 queries. In contrasts, the CodeMatcher ran fast and decreased the code search time by 98.48% from 695.03s to 10.50s (i.e., 0.21s per query). It did not need a long time training as DeepCS, and it only required 23.46h to preprocess code (23.18h for code parsing and 0.28h for code indexing) as DeepCS.

Additionally, the 23.46 hours of code preprocessing seems time-consuming for CodeMatcher. However, there are about 17 million methods, as shown in Table 4, and each method only takes about 0.005s for code preprocessing on average. Thus, CodeMatcher can support the dynamic and rapidly expanded code scale of GitHub as this model requires no optimization, where changed or new methods can be rapidly parsed and indexed. In contrasts, the DeepCS model cannot support these features entirely because it needs to be optimized frequently for its randomness and overfitting issues referred to in Section 4, and the optimized model should vectorize the method again, taking at least 0.5h in our scale of codebase.

TABLE 6: Time efficiency comparison between CodeMatcher and DeepCSe in three different phases.

Model	Train	Preprocess	Search
DeepCSe	58.16h	24.51h	695.03s
CodeMatcher	-	23.46h	10.50s

Sensitivity Analysis. For CodeMatcher, S_{name} and S_{body} are two matching scores to finally determine the ranks of searched results, but it is unknown how much they attributed to the performance of CodeMatcher. Thus, we performed a sensitivity analysis on CodeMatcher by removing S_{body}. Table 3 shows that the model only using S_{name} (CodeMatcher-S_{body}) obtains MRR=0.68, Recall@1/5/10 = 0.6/0.72/0.74, and Precision@1/5/10 = 0.6/0.53/0.5 respectively. Comparing with CodeMatcher, MRR reduced by 4.23%, where Recall@/1/5/10 slided by 6.25%/5.26%/2.63% separately while Precision@/1/5/10 decreased by 6.5%/8.28%/11.58% respectively.

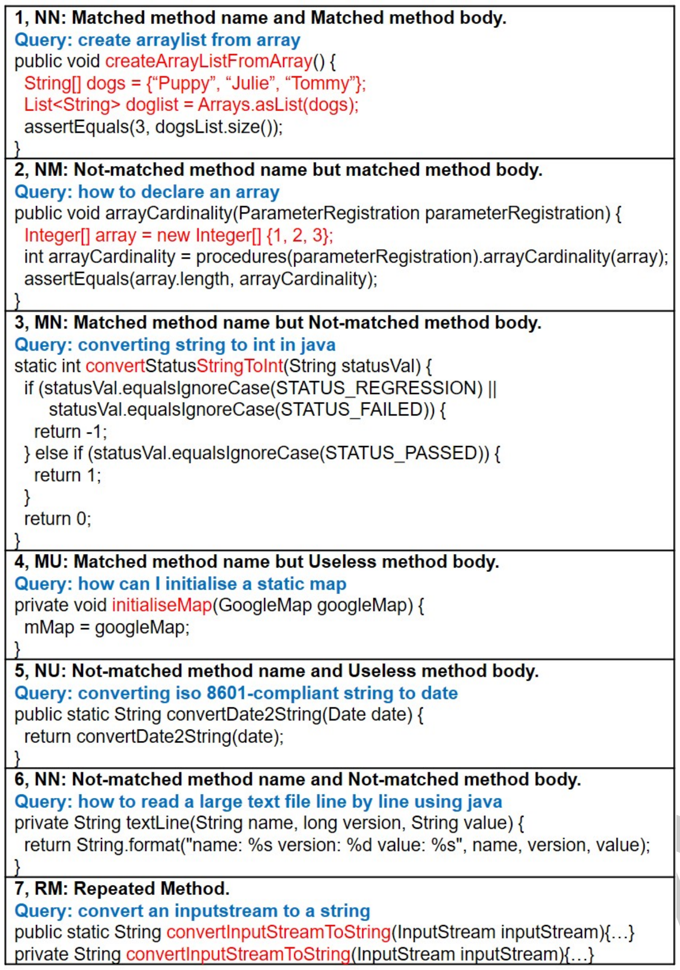
These results indicate that the score S_{name} that matches query keywords with method name dominated the performance of CodeMatcher, and imply that method name is a significant bridge for the semantic gap between query and code. Furthermore, although the influence of S_{body} that match query keywords with method body is low, we cannot ignore its contribution, especially for recommending code methods to developers with more JDK APIs.

Result 2: *A simpler and faster model CodeMatcher that matches query keywords with method code can outperform DeepCS, where MRR improved by 97% and the searching time is over 66 times faster.*

6 Discussion

6.1 What are the Pros and Cons of CodeMatcher vs. DeepCS?

To compare CodeMatcher and DeepCS, we classified their 500 returned methods (50 queries $\times$ top-10 results) into



1, NM: 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);
  assertEquals(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 initialise 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){...}
```

Fig. 4: Definitions of 7 categories on search results and their query-code examples.

seven categories in view of code matching. Note that we chose the best results of DeepCS for analyzing and comparison. Fig. 4 illustrates the definitions and real query-code examples for each category. To be specific, the categories identified whether keywords in query match a code's method name or body; 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 7 lists the classification results for different models.

The reasons why DeepCSr succeeded and failed. From the Table 7, we observed that the re-ran model DeepCSr$_{best}$ tested on original codebase obtained a 41.8% success (MM and NM), where 36.4% of success in MM was due to a correct semantic matching between query and method, as Fig. 4(1); the 5.4% of success in NM implies that DeepCSr$_{best}$ 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 Fig. 4(2).

However, there are 58.2% of failed results, where 1% of failures were caused by returning repeated methods (RM). The source code provided by Gu et al. [12] 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 with some negligible difference, e.g., the modifier difference, as shown in Fig. 4(7). Meanwhile, 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 Fig. 4(3).

For the 10.4% of failures (MU and NU), we found that DeepCSr$_{best}$ 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 Fig.4(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.

For the most part (42.8%) of features (NN), DeepCSr$_{best}$ completely mismatched the code to queries, as illustrated in Fig 4(6). We attribute these failures to the insufficient model training, because (1) DeepCS was optimized by pairs of method and Javadoc comment, but 500 epochs of training with randomly selected pairs cannot guarantee its sufficiency; (2) DeepCS 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 stop because of the convergence of its loss function values.

DeepCSe vs. DeepCSr. As referred to in Section 4.2, the DeepCS model running on new codebase (DeepCSe$_{best}$) suffered from overfitting. Table 7 shows that comparing with DeepCSr, the success rate of DeepCSe$_{best}$ slid from 41.8% to 21% in terms of MM and NM, and 73% of the failures come from the useless methods (MU and NU) and unmatched methods (NN). 30% of failures in returned useless methods also imply the importance of self-contained methods implemented by full of JDK APIs. And the rest 43% of totally unmatched methods suggest that the out-of-vocabulary tokens are meaningful for code search and the limited size of vocabulary may confine the extendability of the DeepCS model.

CodeMatcher vs. DeepCS. For the proposed model CodeMatcher that matches keywords in query with a method, Table 7 shows that 57% of code search succeeded due to 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 (6.4%) and NN (25%). The 11% 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. Same as DeepCS models, CodeMatcher also returned 0.6% repeated methods. Thus, filtering redundant methods by their MD5 hash values, as referred to in Section 5.1, is not enough. A better choice would be comparing the API usages, data structure, and working flow in the method body.

Comparing with DeepCS models, the main advantage of CodeMatcher is the correct keywords matching between query and code, i.e., a high percentage of MM. However,

TABLE 7: Classification of 500 code search results into 7 categories for different models.

Type	Description	DeepCSr$_{best}$	DeepCSe$_{best}$	CodeMatcher	GitHub Searcher
MM	Matched method name and Matched method body.	182 (36.40%)	93 (18.60%)	285 (57.00%)	78 (15.60%)
NM	Not-matched method name but Matched method body.	27 (05.40%)	12 (02.40%)	0 (00.00%)	4 (00.80%)
MN	Matched method name but Not-matched method body.	20 (04.00%)	25 (05.00%)	32 (06.40%)	40 (08.00%)
MU	Matched method name but Useless method body	19 (03.80%)	53 (10.60%)	27 (05.40%)	40 (08.00%)
NU	Not-matched method name and Useless method body.	33 (06.60%)	97 (19.40%)	28 (05.60%)	5 (01.00%)
NN	Not-matched method name and Not-matched method body.	219 (42.80%)	215 (43.00%)	125 (25.00%)	182 (36.40%)
RM	Repeated Method.	5 (01.00%)	2 (00.40%)	3 (00.60%)	151 (30.20%)

CodeMatcher cannot handle partial matching only on code body (i.e., NM=0). But this is what DeepCS models are good at (33 NM for DeepCSr) because it can capture high-level intent between query and method by joint embedding. But DeepCS model suffers from the out-of-vocabulary issue, where the number of NM for DeepCSe drops much. Therefore, CodeMatcher's advantages are good to compensate for DeepCS' disadvantages.

6.2 How Far is CodeMatcher from Optimum?

From Table 1, we found that CodeMatcher cannot return any correct results (i.e., NF, not-found) for 12 queries. We assumed that this is due to two reasons: *(1) mislabeled metadata,* we observed that 12.65% of token property was mislabeled by the used Stanford parser, which frequently identified the 'String' token as a verb instead of a noun; *(2) oversimple algorithm,* the keywords generation algorithm and the reranking algorithm in CodeMatcher are too simple to capture a semantics of query with complex syntax. To verify this assumption, we did the following experiments.

We manually comprehended 12 NF queries and searched function names by combining query tokens in our codebase. Results in Table 8 show that 3 NF queries can be correctly extracted, including 'createStringFromFile', 'readFileLineByLine', and 'appendTextToFile'. These results indicate that improving our keyword generation algorithm can enhance the MRR by 7.04% from 0.71 to 0.76.

When we perform code search on GitHub with manually generated keywords in Table 8, nearly all NF queries can find correct methods, except for the No.22 query. These results imply that extending codebase will further increase the performance of CodeMatcher in terms of MRR by 19.74% from 0.76 to 0.91.

For the No.22 query *"iterating through a collection, avoiding concurrent modification exception when removing in loop"*, we cannot find code in GitHub with method name 'iterateConcurrentCollection'. However, we found a class with the same name, which implemented the 'iterate' part with a function named 'updateThreadList' and implemented the 'iterate collection' part in a main function, respectively. Therefore, the method-level code search cannot satisfy all searching cases as a high-level specification should be done with a collaboration of multiple methods. When considering this class-level search, the MRR can boost from 0.91 to 0.92.

Besides, the above results indicate that all NF queries can be found by combining query tokens on GitHub-scale codebase. Thus, the rest of the MRR improvements from 0.92 to the optimum 1 could be gained by optimizing the CodeMatcher's reranking algorithm.

TABLE 8: Results on whether method names can be found in our collected codebase and GitHub, where the 'query ID' in this table matches the No. of 50 queries in Table 1.

Query ID	Function/Class Name	Codebase	GitHub
14	sqrtIsInteger	0	1
16	createStringFromFile	1	1
22	iterateConcurrentCollection	0	0
25	sortObjectList	0	1
27	padZeroOnLeft	0	1
31	atLeastTwo	0	1
40	readFileLineByLine	1	1
42	appendTextToFile	1	1
43	convert8601StringToDate	0	1
46	splitStringWithWhitespace	0	1
48	invokeMethodByString	0	1
49	getPlatformNewline	0	1

6.3 What is the Advantage of CodeMatcher over the GitHub Search Engine?

The GitHub search engine, also an IR-based model, is what developers frequently used for code search in the real world. Comparing CodeMatcher with GitHub searcher is helpful for understanding the usefulness of CodeMatcher. To investigate the advantages of CodeMatcher over GitHub searcher, we used the 50 queries in Table 1 as their inputs. However, we need to note that GitHub searcher is different from CodeMatcher in three aspects: *(1) Larger-scale codebase.* As GitHub cannot control the search scope as our experiment setup for code snippets (only for repositories), its codebase contains more repositories. *(2) Wider context.* GitHub searcher 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 searcher returns a list of code snippets, but they would not necessarily be a method as what CodeMatcher returns.

During the code search of GitHub searcher, it returns a list of code snippets with matched keywords in highlights. As the code snippet could be a class, method, comment, etc., to make a fair comparison, we inspect the method related to the returned code snippet. However, during the code inspection, we found that many correct code snippets are returned just because the query matched a Stack Overflow link with same query words, such as the link *"http://stackoverflow.com/questions/309424/read-convert-an-inputstream-to-a-string"*. To cope with this kind of bias, we excluded the code snippets whose matched keywords in highlight are as the same as the query. Finally, we identified the query-code relevancy for the top-10 inspected results.

From the results in Table 3, we can notice that

GitHub searcher gains MRR = 0.44 with Recall@1/5/10 = 0.28/0.6/0.64 and Precision@1/5/10 = 0.28/0.21/0.17. Comparing with CodeMatcher, its MRR deceased by 38.03% from 0.71 to 0.44, where Recall@/1/5/10 dropped by 56.25%/21.05%/15.79% and Precision@1/5/10 slid by 56.25%/63.79%/70.18% respectively. By analyzing the classification of these search results in Table 7, we can observe that GitHub searcher shows poorer performance not mainly because of its larger-scale codebase. One main reason for the failures is the unmatched keywords (36.4% for NN) since GitHub searcher cannot support natural language based query search. The other reason is that it does not exclude repeated methods (30.2% for RM). Moreover, except for the disadvantages of GitHub searcher, we can still learn something from it. Specifically, the FRanks in Table 1 show that it can obtain correct results even though CodeMatcher and DeepCS do not work. By inspecting the correct code snippets, we found that GitHub searcher works in these cases because the query matches the comments or Javadoc around a method. Therefore, it would be promising to consider the connection between query and comment/Javadoc for further improving CodeMatcher and DeepCS.

6.4 Implication: Occam's Razor

Our study shows that it is worth simplifying complex approaches or trying simple and fast methods on software engineering tasks, same as the recommendation of the well-known Occam's razor principle [28]. The main advantages of simplified approaches are: *(1) easily explained*, satisfy intuition and present a generalizable white-box method (e.g., a high-quality code snippet whose method name matches developers' expectation is likely what they search for); *(2) controllable*, mitigate the impacts of noises (e.g., the noisy labels for model training), randomness (e.g., parameter initialization and randomly selected training data), and unknown boundary (e.g., when should synonyms are beneficial for tokens and which ordered tokens are more important); *(3) easy to re-run*, let model ready to run and avoid expensive efforts before startup, e.g., DeepCS requires 50h for model training and if it is optimized again the parsed codebase should be updated with at least 0.5h every time.

6.5 Threats to Validity

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

Manual Evaluation by Developers at Baidu. The relevancy of returned results to 50 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; and if a confliction occurred, the developers performed an open discussion to resolve it. 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. Queries that fail are identically assigned with a FRank of eleven following Gu et al. [12], which could be biased from the real relevancy of code methods. 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. [12], 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. [12] in a systematic procedure, as referred to in Section 2. In the future, we will extend the scale and scope of the code search queries. 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 languages is easy and applicable.

7 Road Ahead for Code Search

Essentially, the code search aims to bridge the semantic gap between natural language and programming language, no matter for the deep-learning-based model DeepCS or the simplified model CodeMatcher. Therefore, this section analyzes the road ahead for code search under the view of linguistics between two languages with three related aspects: lexicon, syntax, semantics, and pragmatics.

Lexicon, the used vocabulary. DeepCS fixed its vocabulary size to 10,000 so that it will miss important new words from the frequently updated codebase and varied queries, and regarded new words as noises. It mitigates this issue by two aspects: stemming words from prefix and suffix; addressing synonyms by the joint embedding technique. However, their capabilities are limited for the large-scale new words from a dynamic codebase like GitHub. Also, we observed that the semantics of programming entity (e.g., File) in a query should be unique, but DeepCS may return code named with synonyms, leading to wrong search results.

To solve this problem, the proposed IR-based model CodeMatcher performed a fuzzy match between stemmed keywords in query and code. In this way, the whole indexed codebase can be regarded as vocabulary, and modifying codebase is equivalent to updating vocabulary. Besides, the fuzzy match can handle the compound issue. For example, the words "input stream" in query can match the compound word "inputstream" in the method. Meanwhile, CodeMatcher only generated synonyms for the verbs in queries that were not found in the codebase to control the usage of synonyms. But CodeMatcher also suffers from two lexical cases: *(1) abbreviation*, simplifying 'initialize' to 'ini'; *(2) acronym*, using the 'RMSD' to stand for the "root mean square deviation". Meanwhile, abbreviations and acronyms frequently occur in code. Thus it is worth considering more kinds of lexical features on words.

Suggestion 1: *Considering the importance of lexical factors in code search for the dynamic large-scale codebase, including new words, compound words, synonym, abbreviation, and acronym; be careful to the uniqueness of programming entities in the query.*

Semantics via syntax, the meaning in language expressed by the structure of sentences. DeepCS leveraged RNN to embed query and code, assuming that their semantics are represented by the sequential relationship between words. However, we observed that for the query *"convert int to string"* DeepCS returned many methods named by 'convertStringToInt'. In this case, DeepCS only learned the order of 'convert' and 'to' but missed the more important order of 'int', 'to', and 'string'.

To conquer this representative issue, CodeMatcher assigned higher weights on programming entities 'String' and 'Int', and reranking the returned methods whose name matched query keywords in order. Although this rule is straightforward and useful, it cannot capture high-level intent in query and code like DeepCS. Therefore, a better choice is to combine CodeMatcher with DeepCS. Moreover, both models cannot handle complex query like "fastest way to determine if an integer's square root is an integer", so the syntax structure of query should be further considered carefully.

Suggestion 2: *Balancing the pros and cons of CodeMatcher and DeepCS; paying more attention to the order of programming entities; considering the queries with complex syntax structure.*

Pragmatics, the way in which context contributes to meaning. DeepCS represented the semantics in code by a tuple of <*method name, API sequence, and token*>, where APIs include constructor invocation and methods calls in the method body. This representation means that DeepCS divided the method body into API sequence and a set of tokens with bag-of-words assumption. However, DeepCS missed the semantic integrity of code in four aspects: *(1) parameters,* it excluded the method parameters and returned type, which contributes much semantics for code; *(2) programming entities,* the tokens in method body are just instances of programming types, which can be varied for different context; *(3) token order,* the sequence of token in code should be meaningful, but the bag-of-words assumption missed this semantics and just treated it as common text; *(4) divided context,* in the code, API sequence and tokens are sequentially mixed, so that the divided representation for method body could lose semantics in context.

To better maintain the semantic integrity and reduce the variation of customized context in code, CodeMatcher combined API sequence with token and represented as a sequence of fully qualified tokens, as referred to in Section 5.1. However, CodeMatcher still has two issues: *(1) divided context,* it still divided a code context into method name and the other, and assign much higher weights on method name; *(2) missed context,* it missed the code context in related comment and Javadoc, and Section 6.3 approved their importance for code search; *(3) higher-level context,* Section 6.2 showed that a code search maybe not method-level but class-level or even higher; *(4) structural context,* it is lucky that CodeMatcher found the code for the query "breaking out of nested loops in java", but method name usually reflects its functional specification, not its structural information. Therefore, considering the structural context in code would be helpful for code search.

Suggestion 3: *Considering the semantic integrity of a code context; paying more attention to higher-level context (e.g., class-level) and structural context for code search.*

To sum up, matching query tokens with method name and body strings as CodeMatcher is not enough for the code search task, especially for the queries with complex syntax structures. For the real-world code search, we just move one step further. For the road ahead, we suggest to combine the deep-learning-based model and the IR-based model and balance their pros and cons.

8 Related Work

Categories of Code Search. In the software development, developers may directly search existing applications to work on [11], and many application search engines have been built [15], [16], [29], [30], [31], [32]. 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) [11]. For this reason, method-level code search has been studied for decades [33], [34], [35], [36], 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 [2], [6], [11], [27], [37], [38], and build a model to fill the semantic gap between natural language (i.e., search query) and programming language (i.e., method source code) [12], [13], [14], [20], [39]. Better code search techniques can boost the rapid software development [3] and promote other search-based researches, such as program synthesis [40], [41], code completion [42], [43], [44], program repair [45], [46], and mining software knowledge [47].

For the setting of method-level code search, this study works on the 'query-codebase' search following previous studies [12], [13]. 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], [48], [49]. 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 [12], [13], [14], although there are some useful code provided in Stack Overflow [50] or software development tutorials [51].

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 [17]. Google Code Search[20], Koders[21] and Krugle[22] were few promising systems [14].

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 [9], [18], [19], [20], [21]. For example, Hill et al. [18]

20. https://code.google.com/
21. www.koders.com
22. www.krugle.com

rephrased queries according to the context and semantic role of query words within the method signature. Haiduc et al. [19] proposed Refoqus that reformulated query by choice of reformulation strategies, and a machine learning model recommends which strategy to use based on query properties. Lu et al. [20] extended a query with synonyms generated from WordNet [52]. McMillan et al. [9] proposed Portfolio, which returns a chain of functions through keyword matching and speeds up search by PageRank [53]. All query processing models can outperform early search engines, such as the substantial improvements of Portfolio over Google Code Search and Koders [9].

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 [14]. To push the code search study forward, Bajracharya et al. [14] 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. [39] proposed RACS, a search framework for JavaScript that considers API relationships, including their sequencing, condition, callback relationships, etc. Meanwhile, Lv et al. [13] 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. [54] proposed a model to expand query with semantically related API class names and search the best-matched source code.

Recently, Gu et al. [12] 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 [14] and CodeHow [13]. They attributed these improvements to the successful application of 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. As described in Section 5.1, 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 in searching accuracy and time efficiency.

9 CONCLUSION

The challenge for code search is how to fill the semantic gap between a query in natural language and code in programming language. Recently, Gu et al. proposed a state-of-the-art model DeepCS that leverages deep learning approaches 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, and the rationale of DeepCS was not fully investigated.

In this paper, we first re-ran Gu et al.'s model with the default settings, and analyzed its performance on our newly collected larger-scale codebase. Experimental results show that DeepCS is unstable due to its intrinsic randomness. DeepCS may not work well for dynamic codebases that grow over time like GitHub for its overfitting issue. Inspired by the features of DeepCS, we proposed a simple and faster model that leverages the information retrieval technique to simplify DeepCS. Experimental results show that CodeMatcher is over 66 times faster and it substantially outperforms DeepCS on a larger-scale codebase by 97% in terms of MRR. Finally, we conducted an in-depth analysis of two models and provided some suggestions for the road ahead on code search.

ACKNOWLEDGMENTS

This research was supported by the National Key Research and Development Program of China (2018YFB1003904).

REFERENCES

[1] J. Singer, T. Lethbridge, N. Vinson, and N. Anquetil, "An examination of software engineering work practices," in *CASCON First Decade High Impact Papers*. IBM Corp., 2010, pp. 174–188.

[2] C. Sadowski, K. T. Stolee, and S. Elbaum, "How developers search for code: a case study," in *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*. ACM, 2015, pp. 191–201.

[3] C. McMillan, N. Hariri, D. Poshyvanyk, J. Cleland-Huang, and B. Mobasher, "Recommending source code for use in rapid software prototypes," in *Proceedings of the 34th International Conference on Software Engineering*. IEEE Press, 2012, pp. 848–858.

[4] J. Brandt, P. J. Guo, J. Lewenstein, M. Dontcheva, and S. R. Klemmer, "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, 2009, pp. 1589–1598.

[5] J. Brandt, M. Dontcheva, M. Weskamp, and S. R. Klemmer, "Example-centric programming: integrating web search into the development environment," in *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*. ACM, 2010, pp. 513–522.

[6] H. Li, Z. Xing, X. Peng, and W. Zhao, "What help do developers seek, when and how?" in *2013 20th Working Conference on Reverse Engineering (WCRE)*. IEEE, 2013, pp. 142–151.

[7] 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)*, vol. 21, no. 1, p. 4, 2011.

[8] K. T. Stolee, S. Elbaum, and D. Dobos, "Solving the search for source code," *ACM Transactions on Software Engineering and Methodology (TOSEM)*, vol. 23, no. 3, p. 26, 2014.

[9] C. McMillan, M. Grechanik, D. Poshyvanyk, Q. Xie, and C. Fu, "Portfolio: finding relevant functions and their usage," in *Proceedings of the 33rd International Conference on Software Engineering*. ACM, 2011, pp. 111–120.

[10] M. Gharehyazie, B. Ray, and V. Filkov, "Some from here, some from there: Cross-project code reuse in github," in *2017 IEEE/ACM 14th International Conference on Mining Software Repositories (MSR)*. IEEE, 2017, pp. 291–301.

[11] S. K. Bajracharya and C. V. Lopes, "Analyzing and mining a code search engine usage log," *Empirical Software Engineering*, vol. 17, no. 4-5, pp. 424–466, 2012.

[12] X. Gu, H. Zhang, and S. Kim, "Deep code search," in *2018 IEEE/ACM 40th International Conference on Software Engineering (ICSE)*. IEEE, 2018, pp. 933–944.

[13] 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 *2015 30th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 2015, pp. 260–270.

[14] S. Bajracharya, T. Ngo, E. Linstead, Y. Dou, P. Rigor, P. Baldi, and C. Lopes, "Sourcerer: a search engine for open source code supporting structure-based search," in *Companion to the 21st ACM SIGPLAN symposium on Object-oriented programming systems, languages, and applications*. ACM, 2006, pp. 681–682.

[15] M. Grechanik, C. Fu, Q. Xie, C. McMillan, D. Poshyvanyk, and C. Cumby, "A search engine for finding highly relevant applications," in *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering-Volume 1*. ACM, 2010, pp. 475–484.

[16] M. Robillard, R. Walker, and T. Zimmermann, "Recommendation systems for software engineering," *IEEE software*, vol. 27, no. 4, pp. 80–86, 2009.

[17] K. Krugler, "Krugle code search architecture," in *Finding Source Code on the Web for Remix and Reuse*. Springer, 2013, pp. 103–120.

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

[19] S. Haiduc, G. Bavota, A. Marcus, R. Oliveto, A. De Lucia, and T. Menzies, "Automatic query reformulations for text retrieval in software engineering," in *Proceedings of the 2013 International Conference on Software Engineering*. IEEE Press, 2013, pp. 842–851.

[20] M. Lu, X. Sun, S. Wang, D. Lo, and Y. Duan, "Query expansion via wordnet for effective code search," in *2015 IEEE 22nd International Conference on Software Analysis, Evolution, and Reengineering (SANER)*. IEEE, 2015, pp. 545–549.

[21] L. Ponzanelli, G. Bavota, M. Di Penta, R. Oliveto, and M. Lanza, "Mining stackoverflow to turn the ide into a self-confident programming prompter," in *Proceedings of the 11th Working Conference on Mining Software Repositories*. ACM, 2014, pp. 102–111.

[22] I. Sutskever, J. Martens, G. Dahl, and G. Hinton, "On the importance of initialization and momentum in deep learning," in *International conference on machine learning*, 2013, pp. 1139–1147.

[23] X. Glorot and Y. Bengio, "Understanding the difficulty of training deep feedforward neural networks," in *Proceedings of the thirteenth international conference on artificial intelligence and statistics*, 2010, pp. 249–256.

[24] M. Li, T. Zhang, Y. Chen, and A. J. Smola, "Efficient mini-batch training for stochastic optimization," in *Proceedings of the 20th ACM SIGKDD international conference on Knowledge discovery and data mining*. ACM, 2014, pp. 661–670.

[25] M. Raghothaman, Y. Wei, and Y. Hamadi, "Swim: Synthesizing what i mean-code search and idiomatic snippet synthesis," in *2016 IEEE/ACM 38th International Conference on Software Engineering (ICSE)*. IEEE, 2016, pp. 357–367.

[26] X. Ye, R. Bunescu, and C. Liu, "Learning to rank relevant files for bug reports using domain knowledge," in *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*. ACM, 2014, pp. 689–699.

[27] I. Keivanloo, J. Rilling, and Y. Zou, "Spotting working code examples," in *Proceedings of the 36th International Conference on Software Engineering*. ACM, 2014, pp. 664–675.

[28] A. Blumer, A. Ehrenfeucht, D. Haussler, and M. K. Warmuth, "Occam's razor," *Information processing letters*, vol. 24, no. 6, pp. 377–380, 1987.

[29] C. Liu, D. Yang, X. Zhang, B. Ray, and M. M. Rahman, "Recommending github projects for developer onboarding," *IEEE Access*, vol. 6, pp. 52 082–52 094, 2018.

[30] C. McMillan, M. Grechanik, and D. Poshyvanyk, "Detecting similar software applications," in *Proceedings of the 34th International Conference on Software Engineering*. IEEE Press, 2012, pp. 364–374.

[31] Y. Zhang, D. Lo, P. S. Kochhar, X. Xia, Q. Li, and J. Sun, "Detecting similar repositories on github," in *2017 IEEE 24th International Conference on Software Analysis, Evolution and Reengineering (SANER)*. IEEE, 2017, pp. 13–23.

[32] C. Liu, D. Yang, X. Zhang, H. Hu, J. Barson, and B. Ray, "Poster: A recommender system for developer onboarding," in *2018 IEEE/ACM 40th International Conference on Software Engineering: Companion (ICSE-Companion)*. IEEE, 2018, pp. 319–320.

[33] S. P. Reiss, "Semantics-based code search," in *Proceedings of the 31st International Conference on Software Engineering*. IEEE Computer Society, 2009, pp. 243–253.

[34] D. Lucredio, A. F. d. Prado, and E. S. de Almeida, "A survey on software components search and retrieval," in *Proceedings. 30th Euromicro Conference, 2004.* IEEE, 2004, pp. 152–159.

[35] A. Mili, R. Mili, and R. T. Mittermeir, "A survey of software reuse libraries," *Annals of Software Engineering*, vol. 5, no. 1, pp. 349–414, 1998.

[36] H. Mili, F. Mili, and A. Mili, "Reusing software: Issues and research directions," *IEEE transactions on Software Engineering*, vol. 21, no. 6, pp. 528–562, 1995.

[37] 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*, vol. 22, no. 6, pp. 3149–3185, 2017.

[38] M. M. Rahman, J. Barson, S. Paul, J. Kayani, F. A. Lois, S. F. Quezada, C. Parnin, K. T. Stolee, and B. Ray, "Evaluating how developers use general-purpose web-search for code retrieval," in *Proceedings of the 15th International Conference on Mining Software Repositories*. ACM, 2018, pp. 465–475.

[39] X. Li, Z. Wang, Q. Wang, S. Yan, T. Xie, and H. Mei, "Relationship-aware code search for javascript frameworks," in *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*. ACM, 2016, pp. 690–701.

[40] J. Galenson, P. Reames, R. Bodik, B. Hartmann, and K. Sen, "Codehint: Dynamic and interactive synthesis of code snippets," in *Proceedings of the 36th International Conference on Software Engineering*. ACM, 2014, pp. 653–663.

[41] T. Gvero and V. Kuncak, "Interactive synthesis using free-form queries," in *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*, vol. 2. IEEE, 2015, pp. 689–692.

[42] V. Raychev, M. Vechev, and E. Yahav, "Code completion with statistical language models," in *Acm Sigplan Notices*, vol. 49, no. 6. ACM, 2014, pp. 419–428.

[43] A. T. Nguyen, T. T. Nguyen, H. A. Nguyen, A. Tamrawi, H. V. Nguyen, J. Al-Kofahi, and T. N. Nguyen, "Graph-based pattern-oriented, context-sensitive source code completion," in *2012 34th International Conference on Software Engineering (ICSE)*. IEEE, 2012, pp. 69–79.

[44] R. Robbes and M. Lanza, "Improving code completion with program history," *Automated Software Engineering*, vol. 17, no. 2, pp. 181–212, 2010.

[45] X. B. D. Le, D. Lo, and C. Le Goues, "History driven program repair," in *2016 IEEE 23rd International Conference on Software Analysis, Evolution, and Reengineering (SANER)*, vol. 1. IEEE, 2016, pp. 213–224.

[46] Y. Ke, K. T. Stolee, C. Le Goues, and Y. Brun, "Repairing programs with semantic code search (t)," in *2015 30th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 2015, pp. 295–306.

[47] M. Gabel and Z. Su, "A study of the uniqueness of source code," in *Proceedings of the eighteenth ACM SIGSOFT international symposium on Foundations of software engineering*. ACM, 2010, pp. 147–156.

[48] O. A. L. Lemos, S. K. Bajracharya, J. Ossher, R. S. Morla, P. C. Masiero, P. Baldi, and C. V. Lopes, "Codegenie: using test-cases to search and reuse source code," in *Proceedings of the twenty-second IEEE/ACM international conference on Automated software engineering*. ACM, 2007, pp. 525–526.

[49] O. A. L. Lemos, S. Bajracharya, J. Ossher, P. C. Masiero, and C. Lopes, "A test-driven approach to code search and its application to the reuse of auxiliary functionality," *Information and Software Technology*, vol. 53, no. 4, pp. 294–306, 2011.

[50] B. A. Campbell and C. Treude, "Nlp2code: Code snippet content assist via natural language tasks," in *2017 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. IEEE, 2017, pp. 628–632.

[51] L. Ponzanelli, G. Bavota, A. Mocci, M. Di Penta, R. Oliveto, M. Hasan, B. Russo, S. Haiduc, and M. Lanza, "Too long; didn't watch!: extracting relevant fragments from software development video tutorials," in *Proceedings of the 38th International Conference on Software Engineering*. ACM, 2016, pp. 261–272.

[52] C. Leacock and M. Chodorow, "Combining local context and wordnet similarity for word sense identification," *WordNet: An electronic lexical database*, vol. 49, no. 2, pp. 265–283, 1998.

[53] L. Page, S. Brin, R. Motwani, and T. Winograd, "The pagerank citation ranking: Bringing order to the web." Stanford InfoLab, Tech. Rep., 1999.

[54] 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*, vol. 44, no. 11, pp. 1070–1082, 2017.