CodeMatcher: A Tool for Large-Scale Code Search Based on Query Semantics Matching

Chao Liu
School of Big Data & Software Engineering, Chongqing University
Chongqing, China
liu.chao@cqu.edu.cn

Xuanlin Bao
School of Big Data & Software Engineering, Chongqing University
Chongqing, China
baoxuanlin@cqu.edu.cn

Xin Xia
Software Engineering Application Technology Lab, Huawei
Hangzhou, China
xin.xia@acm.org

Meng Yan*
School of Big Data & Software Engineering, Chongqing University
Chongqing, China
mengy@cqu.edu.cn

David Lo
School of Information Systems, Singapore Management University
Singapore
davidlo@smu.edu.sg

Ting Zhang
School of Information Systems, Singapore Management University
Singapore
tingzhang.2019@phdcs.smu.edu.sg

ABSTRACT

Due to the emergence of large-scale codebases, such as GitHub and Gitee, searching and reusing existing code can help developers substantially improve software development productivity. Over the years, many code search tools have been developed. Early tools leveraged the information retrieval (IR) technique to perform an efficient code search for a frequently changed large-scale codebase. However, the search accuracy was low due to the semantic mismatch between query and code. In the recent years, many tools leveraged Deep Learning (DL) technique to address this issue. But the DL-based tools are slow and the search accuracy is unstable.

In this paper, we presented an IR-based tool CodeMatcher, which inherits the advantages of the DL-based tool in query semantics matching. Generally, CodeMatcher builds indexing for a large-scale codebase at first to accelerate the search response time. For a given search query, it addresses irrelevant and noisy words in the query, then retrieves candidate code from the indexed codebase via iterative fuzzy search, and finally reranks the candidates based on two designed measures of semantic matching between query and candidates. We implemented CodeMatcher as a search engine website. To verify the effectiveness of our tool, we evaluated CodeMatcher on 41k+ open-source Java repositories. Experimental results showed that CodeMatcher can achieve an industrial-level response time (0.3s) with a common server with an Intel-i7 CPU. On the search accuracy, CodeMatcher significantly outperforms three state-of-the-art tools (DeepCS, UNIF, and CodeHow) and two online search engines (GitHub search and Google search).

Demo Tool Website: http://www.codematcher.cn
Demo Video: https://youtu.be/Od7xHsZ_RWY

*Corresponding author.

ESEC/FSE '22, November 14–18, 2022, Singapore, Singapore
© 2022 Association for Computing Machinery.
ACM ISBN 978-1-4503-9413-0/22/11…$15.00
https://doi.org/10.1145/3540250.3558935

CCS CONCEPTS

• **Software and its engineering** → *Software maintenance tools.*

KEYWORDS

Code Search, Information Retrieval, Semantics Matching

ACM Reference Format:
Chao Liu, Xuanlin Bao, Xin Xia, Meng Yan, David Lo, and Ting Zhang. 2022. CodeMatcher: A Tool for Large-Scale Code Search Based on Query Semantics Matching. In *Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE '22), November 14–18, 2022, Singapore, Singapore.* ACM, New York, NY, USA, 5 pages. https://doi.org/10.1145/3540250.3558935

1 INTRODUCTION

When faced with programming problems, developers favor learning from existing code and finding solutions through code search [4, 21, 28, 29]. This is because code search can significantly improve their development efficiency. During software development, it was observed that more than 90% of developers' code search efforts are used to find code snippets [3], thus this study focuses on searching code methods (i.e., the frequently searched reusable code snippets in Java) following previous studies [2, 5, 24]. Moreover, our study focuses on the query with natural language description. The goal of our study is to directly find relevant code methods from a large-scale codebase with the same semantics as the query [22].

Early code search studies leveraged information retrieval (IR) techniques to measure the keywords matching degree between query and candidate code because they are fast [24]. However, the search accuracy of IR-based models is far from satisfactory due to two major issues [16]: *(1) semantic gap*, keywords cannot adequately represent high-level intent implied in queries and the low-level implementation details in code; *(2) representation gap*, query and code are semantically related, but they may be represented by different lexical tokens, synonyms, or language structures.

To address these issues, researchers leveraged deep learning (DL) techniques to embed query and code into a shared high-dimensional vector space. Therefore, the code search can be performed by calculating the cosine similarity between code and query vectors. Two representative models are DeepCS [16] and UNIF [5]. Their major advantages are the capabilities of understanding irrelevant and noisy keywords, capturing sequential relationships between words in query and code, mapping query intent to the semantics of code methods by measuring their semantic similarity [16]. However, the DL-based models need time-consuming training and their response time for code search is slow.

In our prior work [23], to fuse the advantages of IR-based and DL-based models, we proposed an IR-based model CodeMatcher that incorporated the above three features of DL-based models. Generally, CodeMatcher leverages Elasticsearch [14], a Lucene-based text search engine to index codebase and perform fuzzy search with the identified important keywords from search queries. To improve query understanding, CodeMatcher removes noisy keywords according to some collected metadata and replaces irrelevant keywords with synonyms selected from the codebase. To optimize the ranking of code methods searched by Elasticsearch, we reranked the candidate code based on the sequential relationships between words in query and code.

In this paper, we strengthen CodeMatcher by implementing it as a publicly accessible code search engine website. Using CodeMatcher, developers can use natural language to describe a Java programming task as a query. For a query request, CodeMatcher outputs the top-10 Java code methods whose programming semantics highly related to the intent of query. To help other researchers replicate and extend our work in the future, we have publicly released the **replication package [20]** of this tool demonstration.

To evaluate the performance of CodeMatcher, we compare it with three baseline models (i.e. DeepCS [16], CodeHow [24], and UNIF [5]) on a large-scale codebase with 16+ million Java methods and 174 real-world queries. Experimental results showed that CodeMatcher achieves an MRR of 0.60, outperforming three baselines by at least 46.3%. In terms of query processing efficiency, it only takes 0.3s for code search per query, 8 times faster than CodeHow while 1.2K+ times faster than DeepCS and UNIF. Moreover, further experiments showed that CodeMatcher outperforms two online search engines, i.e., GitHub and Google search, by 33.3+% in terms of MRR. These above results demonstrated the effectiveness, time-efficiency, and usefulness of our tool.

2 APPROACH

Fig. 1 illustrates the overall framework of CodeMatcher: 1) the first phase preprocesses a large-scale codebase and builds indexing for it; 2) the second phase performs code search in three steps, including query understanding, iterative fuzzy search, and reranking.

2.1 Phase-I: Codebase Preprocessing and Indexing

To build codebase for code search, the first phase extracts source code files from Java repositories and parsed each file into an abstract syntax tree (AST). By traversing the AST, we obtained name and body of each code method. Besides, the method body is parsed as a sequence of fully qualified tokens (e.g., converting *String* to *java.lang.String*) in method parameters, method body, and returned type. After retrieving pieces of information (i.e., method name, parsed method body, and source code), we input them into Elasticsearch [14] and index the codebase.

Table 1: Five word importance levels for programming based on the word property (e.g., verb or noun) and whether the word is a class name in JDK.

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

2.2 Phase-II: Code Search

In the second phase, CodeMatcher performs code search on the indexed codebase for a query in three steps. The first step extracts some metadata to assist the query understanding, and addresses noisy and irrelevant words in the query. The second step quickly retrieves a set of code methods from codebase for the query based on the collected metadata. The third step optimizes the ranking of the search methods in the previous step by measuring the matching degrees between the query and code methods.

2.2.1 Step-1: Query Understanding. To better understand the core semantics of a query, we address noisy words in the query at first. Specifically, we leveraged the Stanford Parser [6, 15, 30] to identify the parts-of-speech (e.g., verb or noun) of query words, which is called "word property" hereafter. Next, we filtered out three types of noisy words in the query that are rarely used for coding [1] according to the word property: 1) the question words with related auxiliary verbs, e.g., "how do"; 2) the verb-object/adpositional phrase on the programming language, e.g., "in Java"; 3) the words that were not verbs, nouns, adjectives, adverbs, prepositions, or conjunctions. Afterward, we identified the irrelevant words in query by counting the frequency of each word that was occurred in the method name of codebase. If the frequency is zero, we replaced it by the synonyms generated by WordNet [25]. If multiple synonyms were produced, we chose the one with the highest frequency in the codebase. Subsequently, we stemmed [27] the rest of query words to improve their generalizability for code search.

In addition to the words property and frequency, we identified the third metadata named 'importance' for query words before performing code search. The word importance refers to how important a word used for programming. The importance is categorized by five levels as shown in Table 1: JDK noun (i.e., the classes defined in JDK) is the most important one (level 5); verbs and nouns are important (level 4) as they can represent most of semantics in code; adjectives and adverbs are assigned with lower importance (level 3) as they cannot indicate precise meaning without corresponding noun or verb; preposition and conjunction show less importance (level 2); the other (often meaningless) symbols are of the lowest importance (level 1).

2.2.2 Step-2: Iterative Fuzzy Search. The second step takes the preprocessed query words with metadata from the step one as inputs. It retrieves a set of code methods that highly related to the query for quickly narrowing down the search space. We implemented this process as an iterative fuzzy search. Specifically, it builds a regular string [8] with all remaining query words in order as $".*word_1.*\cdots.*word_n.*"$. We performed code search on the

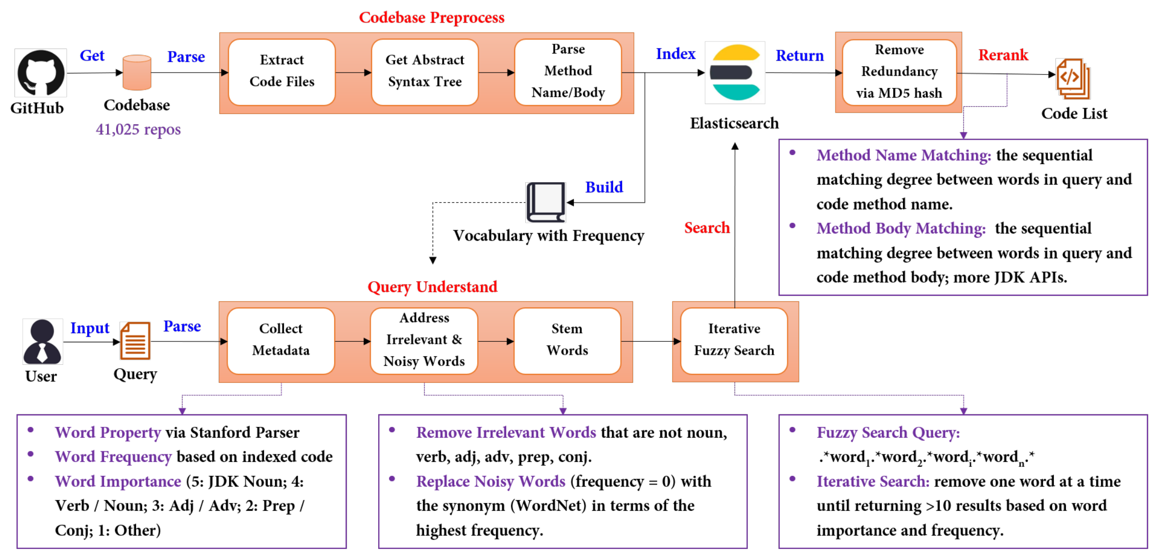

Codebase Preprocess
GitHub
Get
Codebase
41,025 repos
Parse
Extract Code Files
Get Abstract Syntax Tree
Parse Method Name/Body
Index
Elasticsearch
Return
Remove Redundancy via MD5 hash
Rerank
Code List
Method Name Matching: the sequential matching degree between words in query and code method name.
Method Body Matching: the sequential matching degree between words in query and code method body; more JDK APIs.
Build
Vocabulary with Frequency
Search
Query Understand
User
Input
Query
Parse
Collect Metadata
Address Irrelevant & Noisy Words
Stem Words
Iterative Fuzzy Search
Word Property via Stanford Parser
Word Frequency based on indexed code
Word Importance (5: JDK Noun; 4: Verb / Noun; 3: Adj / Adv; 2: Prep / Conj; 1: Other)
Remove Irrelevant Words that are not noun, verb, adj, adv, prep, conj.
Replace Noisy Words (frequency = 0) with the synonym (WordNet) in terms of the highest frequency.
Fuzzy Search Query: .*word1.*word2.*word i.*word n.*
Iterative Search: remove one word at a time until returning >10 results based on word importance and frequency.

Figure 1: Overall framework of CodeMatcher

method names indexed in the codebase. If the total number of returned results is no more than ten, we removed the least important word with lower frequency one at a time according to query metadata, and performed the fuzzy search again until no query words left. For each round of fuzzy search, we filtered out the redundant results by comparing the MD5 hash[9] values of their source code.

2.2.3 Step-3: Reranking. The goal of third step is to rerank the candidate code methods returned from the step two. To reach this goal, we designed a metric (S_{name}) to measure the matching degree between the semantics of query and method name as Eq. (1). Higher value indicates that a higher-ranked method has more overlapped tokens between query and method name in order. If two methods have the same value of S_{name}, we reranked them based on the matching degree S_{body} between the semantics of query and method body as Eq. (2). Different from S_{name}, we added the last term to represent the ratio of JDK APIs in method body, in terms of the fully qualified tokens. Finally, we returned the top-10 reranked results.

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

$$\begin{aligned} S_{body} = & \frac{\#API\ words\ matched\ query\ words}{\#query\ words} \\ & \times \frac{Max[\#API\ words\ orderly\ matched\ query\ words]}{\#query\ words} \\ & \times \frac{\#JDK\ APIs}{\#APIs} \end{aligned} \tag{2}$$

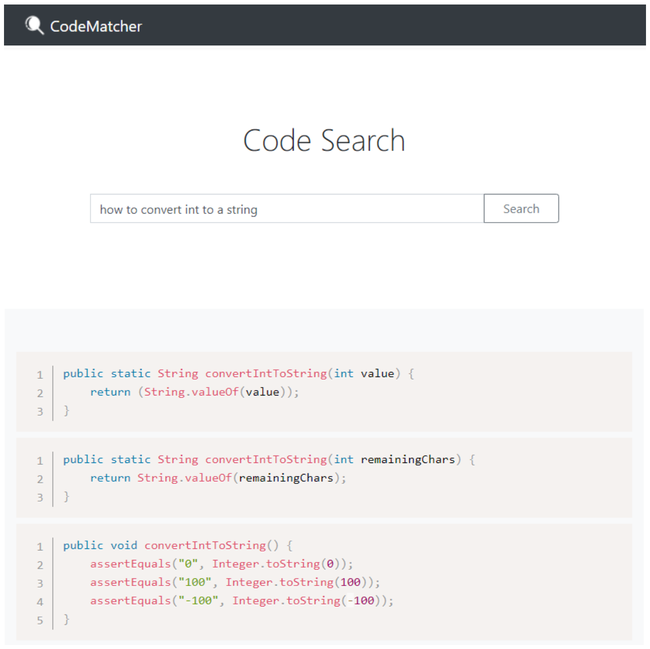
3 TOOL IMPLEMENTATION AND USAGE

We implemented CodeMatcher as a web search engine. The following subsections describe CodeMatcher's codebase, implementation, and usage scenarios.

Data Collection. We chose GitHub as our data source. We searched candidate Java repositories created from July 2016 to December 2018 with more than five stars by using the PyGithub [26] library. This library provides interfaces to call the GitHub APIs[11]. We formed the search string with *"language:java stars:>5 created:2016-07-01..2018-12-31"*. As GitHub only returns the top-1k searched results, we split the time duration (*"2016-07-01..2018-12-31"*) of repository creation into small periods. Afterward, we performed the search in order and extracted the user name and repository name of each repository. In total, we included 41,025 Java repositories. Subsequently, we downloaded their source code according to their user name and repository name.

Data Preprocessing. To build a codebase for code search, we extracted all the Java source code (i.e., "*.java") from our collected repositories. For each source code file, we parsed it into an abstract syntax tree (AST) and obtained method components (i.e., method name, input/output parameters, and method body). Meanwhile, we inferred the fully qualified name of each local variable or class within input/output parameters and method body according to the code context, e.g., transforming *"String"* to *"java.io.String"*. Finally, we obtained 16,611,025 code methods with a total of 70,332,245 lines of code. We implemented the data preprocessing as a tool named Janalyzer [19] and shared it on GitHub.

Codebase Indexing. We leveraged the Elasticsearch [10] with version 8.2 as our basic search engine for codebase. We ran it as a back-end service with command *"./bin/elasticsearch-8.2.0 -d"*. We represented a code method a triple ⟨*method name, parsed method body, source code of method*⟩, where the parsed method body is the sequence of fully qualified names generated from input parameter, method body, and output parameter. We created indexing in Elasticsearch with the following settings: *mappings = "properties": "method": "type": "text", "parsed": "type": "text", "source": "type": "text"*. And then, we filled all these data into it.



```
public static String convertIntToString(int value) {
    return (String.valueOf(value));
}
```

```
public static String convertIntToString(int remainingChars) {
    return String.valueOf(remainingChars);
}
```

```
public void convertIntToString() {
    assertEquals("0", Integer.toString(0));
    assertEquals("100", Integer.toString(100));
    assertEquals("-100", Integer.toString(-100));
}
```

Figure 2: The homepage of CodeMatcher

Code Search Website. We implemented the search engine CodeMatcher based on the Flask [7] web framework. Fig. 2 shows an example of the CodeMatcher website. Specifically, a developer needs to input a natural language query in the search textbox and click the search button. After the server received the request with the search query, it works as the code search process as described in Section 2.2 to retrieve a set of ranked candidate code methods from Elasticsearch, and finally returned the top-10 related results to the front end interface. Soon after, the website formatted each returned code with highlighted keywords to improve the readability of the code, and showed the code list instantly. Note that Fig. 2 only illustrated the top-3 results due to the limited space, where complete results can be found in our tool demonstration video.

Usage Scenarios. This section presents several examples to illustrate how developers would interact with CodeMatcher. For queries like *"how to convert int to a string"*, developers can quickly find the correct answer (i.e., the first method named *"convertIntToString"*) when browsing the name and body of the top-1 returned method code. The other search results provided the four different ways to *"convert int to string"*. As CodeMatcher considered the sequential semantics of important query words, it can distinguish the difference between two queries *"how to convert int to a string"* and *"converting string to int in java"*. Note that this tool demo only illustrates the top-10 search results returned by CodeMatcher, because we assumed that developers would like to review a limited number of results following our prior work [23] and related studies [5, 16].

4 EVALUATION AND USER STUDY

Effectiveness and Time-Efficiency. To evaluate the effectiveness of CodeMatcher, we obtained 174 real-world queries from four related studies [16–18]. Experimental results showed that CodeMatcher achieved an MRR (mean reciprocal rank) of 0.60, substantially outperforming one IR-based model CodeHow by 62.2%, and two DL-based models (DeepCS and UNIF) by 81.8% and 46.3%. In terms of the time-efficiency, CodeMatcher only takes 0.3s to search code from a large-scale codebase for a query on average, 1.2k times faster than two DL-based models and 8 times faster than CodeHow. Also, CodeMatcher needs no time-consuming model training as DL-based models.

Conciseness and Completeness. Besides, we investigated the conciseness and completeness of the searched code. Conciseness indicates the ratio of irrelevant lines to the total lines [18]. And the completeness means the number of addressed tasks divided by the total number of tasks, where the task includes the intent of the search query and other missed statements [18]. Experimental results showed that the average conciseness and completeness of CodeMatcher are 0.61 and 0.33, significantly outperforming the best baseline by 26.5% and 106.25% respectively. These results indicated that CodeMatcher can return easier to read code and the code can better complete the intent of queries.

Usefulness. To further assess the usefulness of CodeMatcher, we compared CodeMatcher with two existing online search engines, including GitHub[12] and Google[13] search. These two search engines performed code search from all Java repositories in GitHub with our collected 174 real-world queries. Experimental results showed that CodeMatcher outperforms GitHub and Google search engines by 46.3% and 33.3% in terms of MRR, respectively. These results implied the usefulness of CodeMatcher in practical usage.

5 CONCLUSION AND FUTURE WORK

In this paper, we present CodeMatcher, a tool implemented as a code search engine website that retrieves a set of code snippets from a large-scale Java codebase according to a query written in natural language. Generally, CodeMatcher leveraged IR techniques to implement the features of DL-based models, to fuse the advantages of IR and DL-based models. Experimental results showed that CodeMatcher can perform code search with industrial-level response time while outperforming two state-of-the-art DL-based models in terms of search accuracy substantially. In the near future, we plan to extend CodeMatcher to support more programming languages (e.g., Python) and provide more customizable search options.

ACKNOWLEDGMENTS

This research / project is supported by by the National Key Research and Development Project (2021YFB1714200), China Postdoctoral Science Foundation (2022M710519), the Postdoc Foundation of Chongqing (2021LY23), the Key Research and Development Program of Zhejiang Province (2021C01014), the National Nature Science Foundation of China (62002034 and U20A20173), and the National Research Foundation, Singapore under its Industry Alignment Fund — Pre-positioning (IAF-PP) Funding Initiative, and the ARC Laureate Fellowship (FL190100035). Any opinions, findings and conclusions or recommendations expressed in this material are those of the author(s) and do not reflect the views of National Research Foundation, Singapore, and Huawei, China. This research is a non-Huawei service achievement.

REFERENCES

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

[2] Sushil Bajracharya, Trung Ngo, Erik Linstead, Yimeng Dou, Paul Rigor, Pierre Baldi, and Cristina Lopes. 2006. Sourcerer: a search engine for open source code supporting structure-based search. In *Companion to the 21st ACM SIGPLAN symposium on Object-oriented programming systems, languages, and applications*. 681–682. doi:10.1145/1176617.1176671.

[3] Sushil Krishna Bajracharya and Cristina Videira Lopes. 2012. Analyzing and mining a code search engine usage log. *Empirical Software Engineering* 17, 4 (2012), 424–466. doi:10.1007/s10664-010-9144-6.

[4] Joel Brandt, Mira Dontcheva, Marcos Weskamp, and Scott R Klemmer. 2010. Example-centric programming: integrating web search into the development environment. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*. 513–522. doi:10.1145/1753326.1753402.

[5] Jose Cambronero, Hongyu Li, Seohyun Kim, Koushik Sen, and Satish Chandra. 2019. When deep learning met code search. In *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 964–974. doi:10.1145/3338906.3340458.

[6] Danqi Chen and Christopher D Manning. 2014. A fast and accurate dependency parser using neural networks. In *Proceedings of the 2014 conference on empirical methods in natural language processing (EMNLP)*. 740–750. doi:10.3115/v1/D14-1082.

[7] Flask Community. 2022. Flask documentation. https://flask.palletsprojects.com/en/2.1.x/. (2022).

[8] Python Community. 2001. Regular string formation in Python. https://docs.python.org/3/library/re.html. (2001).

[9] Python Community. 2020. Hashlib library in Python. https://docs.python.org/2/library/hashlib.html. (2020).

[10] Elastic. 2022. Elasticsearch official website. https://www.elastic.co/downloads/elasticsearch. (2022).

[11] GitHub. 2022. REST API for GitHub. https://docs.github.com/en/rest. (2022).

[12] GitHub. 2022. The source code search engine for GitHub. https://github.com/search. (2022).

[13] Google. 2022. The Google website search engine. https://www.google.com. (2022).

[14] Clinton Gormley and Zachary Tong. 2015. *Elasticsearch: the definitive guide: a distributed real-time search and analytics engine*. " O'Reilly Media, Inc.".

[15] The Stanford NLP Group. 2002. Stanford Parser. https://nlp.stanford.edu/software/lex-parser.shtml. (2002).

[16] Xiaodong Gu, Hongyu Zhang, and Sunghun Kim. 2018. Deep code search. In *2018 IEEE/ACM 40th International Conference on Software Engineering (ICSE)*. IEEE, 933–944. doi:10.1145/3180155.3180167.

[17] Hamel Husain, Ho-Hsiang Wu, Tiferet Gazit, Miltiadis Allamanis, and Marc Brockschmidt. 2019. Codesearchnet challenge: Evaluating the state of semantic code search. *arXiv preprint arXiv:1909.09436* (2019). doi:10.1145/2786805.2786855.

[18] Iman Keivanloo, Juergen Rilling, and Ying Zou. 2014. Spotting working code examples. In *Proceedings of the 36th International Conference on Software Engineering*. 664–675. doi:10.1145/2568225.2568292.

[19] Chao Liu. 2019. Janalyzer. https://github.com/liuchaoss/janalyzer. (2019).

[20] Chao Liu. 2022. Replication package of codematcher demo. https://github.com/liuchaoss/codematcher-demo. (2022).

[21] Chao Liu, Cuiyun Gao, Xin Xia, David Lo, John Grundy, and Xiaohu Yang. 2021. On the Reproducibility and Replicability of Deep Learning in Software Engineering. *ACM Transactions on Software Engineering and Methodology (TOSEM)* 31, 1 (2021), 1–46. doi:10.1145/3477535.

[22] Chao Liu, Xin Xia, David Lo, Cuiyun Gao, Xiaohu Yang, and John Grundy. 2021. Opportunities and challenges in code search tools. *ACM Computing Surveys (CSUR)* 54, 9 (2021), 1–40. doi:10.1145/3480027.

[23] Chao Liu, Xin Xia, David Lo, Zhiwe Liu, Ahmed E Hassan, and Shanping Li. 2021. CodeMatcher: Searching Code Based on Sequential Semantics of Important Query Words. *ACM Transactions on Software Engineering and Methodology (TOSEM)* 31, 1 (2021), 1–37. doi:10.1145/3465403.

[24] Fei Lv, Hongyu Zhang, Jian-guang Lou, Shaowei Wang, Dongmei Zhang, and Jianjun Zhao. 2015. Codehow: Effective code search based on api understanding and extended boolean model (e). In *2015 30th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 260–270. doi:10.1109/ASE.2015.42.

[25] George A Miller. 1998. *WordNet: An electronic lexical database*. MIT press.

[26] PyGitHub. 2021. Python API for GitHub. https://pygithub.readthedocs.io/en/latest/introduction.html. (2021).

[27] PythonProgramming.net. 2015. NLTK Stem tutorial. https://pythonprogramming.net/stemming-nltk-tutorial/. (2015).

[28] Jianhang Shuai, Ling Xu, Chao Liu, Meng Yan, Xin Xia, and Yan Lei. 2020. Improving code search with co-attentive representation learning. In *Proceedings of the 28th International Conference on Program Comprehension*. 196–207. doi:10.1145/3387904.3389269.

[29] Ling Xu, Huanhuan Yang, Chao Liu, Jianhang Shuai, Meng Yan, Yan Lei, and Zhou Xu. 2021. Two-Stage Attention-Based Model for Code Search with Textual and Structural Features. In *2021 IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER)*. IEEE, 342–353. doi:10.1109/SANER50967.2021.00039.

[30] Neng Zhang, Jian Wang, and Yutao Ma. 2017. Mining domain knowledge on service goals from textual service descriptions. *IEEE Transactions on Services Computing* 13, 3 (2017), 488–502. doi:10.1109/TSC.2017.2693147.