SoCeR: A New Source Code Recommendation Technique for Code Reuse

Md Mazharul Islam and Razib Iqbal
Department of Computer Science
Missouri State University
Springfield, Missouri, USA
mdmazharul068@live.missouristate.edu; riqbal@missouristate.edu

Abstract— **Motivated by the idea of reusing existing source code from previous projects within a software company, in this paper, we present a new source code recommendation technique called "SoCeR" to help programmers find relevant implementations or sample code based on software requirement specifications. SoCeR assists programmers to search existing code repositories using natural language query. Our proposed approach summarizes Python code into sentences or phrases to match them against user queries. SoCeR extracts and analyzes the content of the code (such as variables, functions, docstrings, and comments) to generate code summary for each function which is then mapped to the respective functions. For evaluation purposes, we developed a web-based tool for users to enter a textual search query and get the relevant code search results that were most relevant to the query. In SoCeR, users can also upload new code to enrich the code base with tested code. If adopted, then SoCeR will benefit a software company to build a trusted code base enabling large-scale software code reuse.**

Keywords- Code recommendation, code reuse, code search, software code, query reformulation.

I. INTRODUCTION

A software company can reduce development time and cost by reusing source code from previous projects. To reuse source code in private repositories, we have developed SoCeR, a system to collect, analyze, and search Python source code. To improve the means by which reliable code are reused, SoCeR's design is motivated by finding similar source code based on specific software requirements or natural language even before the actual development phase starts. Even though programming by example was found to be intuitive to many developers as per [1] and [2], the existing techniques, such as [3][4], rely on the developers' coding activities to search or recommend the relevant source code. Compared to these techniques, our goal is two-fold – *a)* Present a lightweight code search tool to reuse source code that have been tested and delivered earlier, and *b)* Search relevant code using textual software requirements specifications and/or natural language.

The rest of the paper is organized as follows: an overview of the existing approaches for source code recommendation and code summarization is given in Section II. In Section III, we present our proposed model for source code recommendation followed by a reformulated search query suggestion technique in Section IV. The evaluation of SoCeR is presented in Section V. Finally, in Section VI, we give our concluding remarks.

II. LITERATURE REVIEW

There are plug-ins and search tools, such as Strathcona [3] and PARSEWeb [4], for standard integrated development environments that recommends code fragments based on the code context and the structural details of the developer's activities. Strathcona uses structural context to recommend source code which is helpful for the developers who are using framework for object-oriented programming. Similar to Strathcona, PARSEWeb, which works with Java object-oriented source code, interacts with Google Code Search engine to gather source code and analyzes partial code samples using Abstract Syntax Trees and Directed Acyclic Graphs that can handle control-flow information and method inlining. However, it might not be intuitive for the developers to search for the related frameworks or code in PARSEWeb if they are unsure of the class structure and dependencies.

Code search engines, such as Koder [5] and Krugle [6], use open source repositories to return link results pointing to the actual project files. This requires the developers to read and understand the search results and then look for the possible match for their query. However, these open-source results might lead to vulnerable code in the final product.

Another tool SWIM [7] suggests C# code snippets by translating user queries into existing APIs of interest using the clickthrough data from the Bing search engine. While the SWIM user query does not need to contain specific type names or methods of interest, it heavily depends on the standard libraries and frameworks to find and match the appropriate APIs. Similar to SWIM, MICA [1] and Exampler [2] also uses standard libraries, such as Java Development Kit, to search for API examples for recommending sample code. To effectively use these tools, the developer must have good knowledge of entire software workflow to get the desired outcome.

Sourcerer [8] is an infrastructure for downloading, parsing, storing, and analyzing Internet-scale software corpora, as well as a search engine supporting keyword-based and structure-based

querying. In Sourcerer, the authors applied topic modelling to mine and search Internet-scale software repositories. The code search feature in Sourcerer is built using indexed keys and ranked entities. Query keywords entered by a user are matched against the set of keywords where each key is mapped to a list of entities, and each entity has a rank associated with it. Thus, the list of entities that are matched against the sets of matching keys are returned as search results to the user.

CodeGenie [9] is a tool that uses test cases as an interface for code search. For example, JUnit test classes must be created to define the desired features and then the test cases are used to find the relevant code in the internet repositories. Therefore, the developers first have to translate the software features based on the requirements specification to code representing the test cases before the CodeGenie could be used.

In Example Overflow [10], the authors relied on Stack Overflow for code snippets. They save the code snippet along with related metadata such as the question title, the user rating, etc. in order to find code snippets that may not contain the search query keyword, but the keyword appears in the contextual data. For searching, it uses keyword search based on the Apache Lucene [11] library, which internally uses the term frequency-inverse document frequency (tf-idf) weight [12]. However, in order for Apache Lucene to search properly, we need to define the parameters that need to be analyzed and indexed.

Authors in [13] proposed a system that can generate pseudocode in natural language from source code. It generates pseudocode from every information present in the source code. For example, if the source has a conditional expression like "if x % 5 == 0:" then it be translated into "if x is divisible by 5". However, we are interested in code summary which can describe the main functionality of a function.

Authors in [14] proposed semantic based code search and their main approach is to take a set of candidate solutions, attempt to transform that set into a more appropriate set, check the resultant set against the user's specifications, and then output all solutions. However, this system requires the user to augment or change the specifications based on the output that the original descriptions/query produced.

Another paper published in 2018 [15] proposed a system named GITSEARCH, a code search engine, on top of GitHub and Stack Overflow Q&A data. Their approach leverages common developer questions and the associated expert answers to augment user queries with the relevant source code entities in order to improve the performance of matching relevant code examples within large code repositories. In contrast, we are relying on private repositories without any associated metadata. Similar to [15], the technique proposed in [16] also used crowdsourced data from Stack Overflow for code searching which automatically identifies relevant and specific API classes from Stack Overflow Q&A site for a programming task written as a natural language query, and then reformulates the query for improved code search. Finally, authors in [17] proposed a technique which searches clone of source code instead of natural language search query. To apply their technique any user must have strong background on the code structure to find particular code snippets as the user has to give source code as input rather than user query.

Now, for source code recommendation we need to know what kind of information a source code provides. There are existing works on retrieving information from source codes, e.g. [18] and [19]. Authors in [18] use three supporting technologies to summarize JAVA source code: a Software Word Usage Model (SWUM) to represent program statements as sets of nouns, verbs, and prepositional phrases; a Natural Language (NLG) system to translate the output of SWUM into readable natural language sentences; a PageRank algorithm to sort the functions based on their importance. Similarly, the proposed method in [19] generates summary comments for Java functions which relies on the code statements and descriptive comments. Authors utilized SWUM to capture the occurrences of words in code as well as their linguistic and structural relationships. Therefore, both of these works require a good naming convention for JAVA functions and parameters.

Motivated by Example Overflow to find code snippets that may not contain the search query text, SoCeR uses summary (a.k.a. descriptor) generated from source code without crowdsourced metadata. Unlike Krugle and Koder, our focus is on local/private code repository which can be reused for future developments because SoCeR does not use the open-source results which might lack the quality for reusing them in another project. Also, SoCeR does not depend on developer activity or feedback and it does not require the user to track the code snippets to actual project files. Finally, compared to the machine learning approaches, such as [20], we opted for information retrieval from source code due to the lack of labeled data.

III. Proposed Source Code Recommendation System

In Fig. 1, we show the component diagram representing different modules of SoCeR. It consists of three main modules: Source code preprocessing and validation, Function descriptor generator, and Source code search.

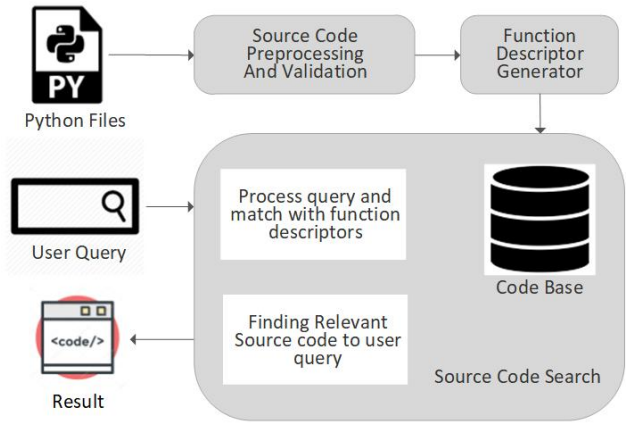

Fig. 1. Complete Workflow of SoCeR.

A. Source code preprocessing and validation

In this version of SoCeR, we considered Python source code for our code base. We used Abstract Syntax Tree (AST) for extracting important attributes like function name and variable names from the source code. An AST is a representation of the abstract syntactic structure of source code where each node

denotes a construct (i.e. variable names, function identifiers etc.) occurring in the source code. In order to make ASTs error free, SoCeR automatically preprocesses the new source code primarily for indentation which otherwise would lead to errors while constructing the AST. For example, in Table I, we give two functions, which are almost identical with the same indentation and function name and body. However, in the *for* loop, the range function [Definition of range function: range(stop), range (start, stop [, step])] was not implemented correctly in Function 2. If we upload both of these functions, then SoCeR will ignore Function-2 due to the syntax error. To correct the indentation, SoCeR uses the pycodestyle [21] utility to determine which parts of the code needs to be formatted first, and then it uses autopep8 [22] Python module to format the code to comply with PEP8 [23] style guide. Finally, SoCeR converts the source code into AST using the Python's ast [24] module so that if there is an error in generating the AST then SoCeR will ignore the code.

B. *Function descriptor generator*

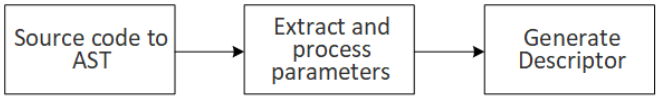

Fig. 2. Workflow of function descriptor generator.

Function descriptor generator generates summary from the source code. The descriptors consist of natural language words, phrases and sentences. Fig. 2 shows the steps associated with the function descriptor generation process. We extract all the functions from the AST first, and then extract the corresponding function names, docstring and variables in order to generate a meaningful summary for each function. Most of the time, readable variable and function names contain important information about the workflow and functionality of the function. Therefore, we processed and considered the meaningful words which can enrich our function descriptor. For example, some variables use camel casing convention, e.g. saveTemperatureValue, based on which we attempt to produce the output "save temperature value". However, if there is a variable "saveTemperatureAsdf", then we will produce the output "save temperature" and ignore the part *Asdf*. Similarly, we process the variable names having underscore ("_") in them, e.g. a variable named get_temperature_value would produce the output "get temperature value". We used wordnet [25] and synset [26] to lookup extracted word from variable names within a dictionary. Finally, we extracted the comments from functions by traversing the function's source code using simple text processing. After extracting the function names, docstring, variable names and comments, we combined these information and generated summary, which is the output of the function descriptor generator as shown in Table II. Now, for faster query search, instead of generating function descriptors for every search query, we store all the uploaded source code's functions along with their descriptors in the database. When a user uploads a source code file using our web interface, SoCeR uses its function descriptor generator to generate the summary of each function of the source code and stores them in the database along with the corresponding function name and body.

TABLE I. COMPARISON BETWEEN VALID AND INVALID FUNCTIONS

Function 1: **Valid**	`def bubbleSort(array):`
	`  n = len(array)-1`
	`  for i in range(0, len(array)):`
	`  for j in range(0, n):`
	`  if array[j] > array[j+1]:`
	`  swap(array, j+1, j)`
	`  n -= 1`

Function 2: **Invalid**	`def bubbleSort(array):`
	`  n = len(array)-1`
	`  for i in range i:`
	`  for j in range(0, n):`
	`  if array[j] > array[j+1]:`
	`  swap(array, j+1, j)`
	`  n -= 1`

TABLE II. SAMPLE FUNCTION BODY WITH ITS DESCRIPTOR

Function	**Descriptor**

`def bubble_sort():`
`  """Implements Bubble Sort"""`
`  n = len(arr)`
`  # Traverse through all array elements`
`  for i in range(n):`
`  for j in range(0, n-i-1):`
`  # traverse the array from 0 to n-i-1`
`  # Swap if the element found is greater`
`  # than the next element`
`  if arr[j] > arr[j+1] :`
`  arr[j], arr[j+1] = arr[j+1], arr[j]`
	Implements Bubble Sort
	bubble sort
	Traverse through all array elements
	traverse the array from 0 to n-i-1
	Swap if the element found is greater than the next element

C. *Source code search*

Referring to Fig. 1, for a user search query consisting of natural English language, SoCeR matches it with the stored function descriptors based on similarity. For calculating the similarity among the function descriptors and user query, we used a text mining technique called term frequency–inverse document frequency (tf-idf). This tf-idf is a statistical measure to evaluate the importance of a word to a document in a collection or corpus. The importance increases proportionally to the number of times a word appears in the document but is offset by the frequency of the word in the corpus.

Typically, the tf-idf weight is composed of two terms - normalized Term Frequency (TF) and Inverse Document Frequency (IDF). TF measures how frequently a term (i.e. word) appears in a document. Since every document is different in length, it is possible that a term would appear much more times in long documents than shorter ones. Thus, the term frequency is often divided by the document length (i.e. the total number of terms in the document) as a way of normalization. In our case, we calculated TF using the following formula:

`TF(t)` = (Number of times term *t* appears in phrases and sentences of descriptors) **/** (Total number of terms in the phrases and sentences of descriptors)

IDF measures how important a term is. While computing TF, all terms are considered equally important. However, it is

known that certain terms, such as "is", "of", and "that", may appear a lot of times but have little importance. Thus, we need to weigh down the frequent terms while scaling up the rare ones. For IDF calculation, we applied the following formula:

```
IDF(t) = log_e(Total number of phrases and sentences of descriptors / Number of phrases and sentences of descriptors with term t in it)
```

We used the Python's TfidfVectorizer [27] module for transforming function descriptors and user query into a matrix of tf-idf features. After calculating the tf-idf weights, we calculated the similarity score between user query and function descriptor to rank the functions based on the cosine similarity. Cosine similarity, as in (1), measures how similar the documents are irrespective of their sizes.

$$cosine\ similarity(d1, d2) = \frac{Dot\ Product(d_1, d_2)}{||d_1|| * ||d_2||} \quad (1)$$

Finally, SoCeR returns the relevant functions according to their similarity scores in response to the user query as output. SoCeR also handles multiple versions of source code. For a new version of a given project, SoCeR stores the source code in addition to the previous version if the new version is different from the previous version. The advantage of keeping all the versions is that if a user performs a search, then SoCeR will recommend the version which will meet user query the most. Because, latest version of a source code might have additional functionality which may be redundant to the user if the user is looking for some basic functionalities.

For nested functions, SoCeR adds function names to the caller function's descriptor which contributes to the similarity score. Therefore, for a given query, SoCeR might also include the caller function in the search result when the search is originally referring to the nested function. For recursive functions, SoCeR similarly adds all the instances of each recursive call to the function descriptor. For multiple branches of decision statements, if there are variables to create multiple branches and they are used multiple times in the function, then SoCeR will store these occurrences in the function descriptor.

IV. QUERY SUGGESTION

During our initial user experiment survey, we observed that users might not get the desired code search results based on their natural language queries due to vocabulary mismatch issues, which is also evident in the literature, such as [28] and [29]. To address this issue, SoCeR offers a feature which suggests queries to the users based on their search text and relevant code descriptors in the repository. For example, if a user is looking for a code snippet that implements a specific sorting algorithm: "merge sort", but entered a generic query, such as "sort items", then SoCeR will recommend relevant code snippets as per Section III.C. for the query "sort items" along with some query suggestions shown in Table III. Now, based on the current code repository, if we use the query "sort items" then SoCeR will return code snippets of bubble sort algorithm and selection sort algorithm as the top two results as shown in Table IV where code snippets for merge sort is ranked in the third position. Therefore, in this example, the user might not get the merge sort algorithm as the top result initially because the query was generic, and the user did not mention the 'merge' keyword in the query. To alleviate this issue, along with the code search results, SoCeR will suggest queries to the users for them to reformulate the original query to get modified code search results. As discussed in Section-III.B, SoCeR generates descriptors for functions that have natural language phrases and sentences. We exploit these descriptors for suggesting queries to the users. For query reformulation, at first, we retrieved the descriptors from the code search results with top similarity scores based on user queries. Then, we considered the phrases and sentences from those descriptors for suggesting them as new queries. To select the phrases or sentences as the candidates for the suggested query list, we applied semantic similarity to the user issued query. For calculating this semantic similarity between user query and the phrases or sentences from the descriptors, we used tf-idf technique which we used earlier in Section-III.C.

TABLE III. QUERY SUGGESTIONS FOR THE QUERY "*SORT ITEMS*"

1. selection sort	4. sorted by value
2. sorted by blue	5. merge sort
3. sorted by red	6. bubble sort

TABLE IV. CODE SEARCH RESULTS FOR THE QUERIES "*SORT ITEMS*" AND "*HOW TO DO MERGE SORT*"

Rank	Results for "**sort items**" query	Results for "**how to do merge sort**" query
1	`def bubbleSort(nlist):` Similarity Score:0.260	`def merge_sort(arr):` Similarity Score:0.449
2	`def selection_sort(arr):` Similarity Score:0.175	`def bubbleSort(nlist):` Similarity Score:0.261
3	`def merge_sort(arr):` Similarity Score:0.175	`def selection_sort(arr):` Similarity Score:0.176

In the literature, there are existing works, such as [30][31][32], where the authors use tf-idf to select the appropriate expansion terms for query reformulation. However, compared to those works, we relied on the source code descriptors only because we do not use user feedbacks and metadata of the source code from StackOverflow or Github. Once the tf-idf is calculated for the user issued query and extracted phrases or sentences, the similarity between them is measured using cosine similarity. From each phrase or sentence, we derived a vector. Then the phrases or sentences are viewed as a set of vectors in a vector space. For example, for the queries "sort items" and "how to do merge sort" the output vector will be:

```
[[0.81480247 0.         0.57973867]
[0.         0.81480247 0.57973867]
```

Then using the cosine similarity, we calculated the similarity between two phrases or sentences. Based on this similarity score, we suggest the phrases or sentences to the users to help them reformulate the original query.

V. EVALUATION

To evaluate SoCeR, we initially populated its code base with 19 Python projects for Internet of Things and Image Processing research with an average of 750 effective lines of code per project from a Software Engineering Capstone class. SoCeR retrieved 583 functions from these projects.

Now, if we enter the following search query: "I want to know how to implement merge sort", then SoCeR will return the top 3 results shown in Table V. In Table V, the first search result is merge_sort with highest similarity score because it had docstring describing merge sort implementation. However, the second and third results are not related to merge sort, but they are sorting algorithms where variables and docstring having the word "sort" are contributing to their respective similarity scores. The second function has slightly better similarity score than the third one because it is smaller in size but the ratio of useful information to its descriptor size is higher.

TABLE V. SAMPLE OUTPUT WITH SIMILARITY SCORES

Ranks	Source Code	Score

1	`def merge_sort(arr):`
	`  """Implementation of Recursive Merge Sort"""`
	`  if len(arr) <= 1:`
	`  return arr`
	`  mid = len(arr) // 2`
	`  left, right = merge_sort(arr[:mid]), merge_sort(arr[mid:])`
	`  return merge(left, right, arr.copy())`
		0.225

2	`def bubbleSort(nlist):`
	`  for passnum in range(len(nlist)-1,0,-1):`
	`  for i in range(passnum):`
	`  if nlist[i]>nlist[i+1]:`
	`  temp = nlist[i]`
	`  nlist[i] = nlist[i+1]`
	`  nlist[i+1] = temp`
		0.150

3	`def selection_sort(arr):`
	`  """Selection sort implementation"""`
	`  for i in range(len(arr)):`
	`  minimum = i`
	`  for j in range(i + 1, len(arr)):`
	`  if arr[j] < arr[minimum]:`
	`  minimum = j`
	`  arr[minimum], arr[i] = arr[i], arr[minimum]`
	`  return arr`
		0.102

While formatted query texts, well documented code, and explicit variable/function names positively impact code search results, in Table VI we show how the availability of different parameter combinations to generate function descriptors affect the source code similarity scores. In Table VII, we present precision data for sample queries to evaluate the effectiveness of SoCeR. Precision represents the fraction of retrieved results that are relevant to the query. In comparison, recall which represents the fraction of the relevant results in the code base is irrelevant in our case because we do not have labeled data as we are working with private code repositories.

We also conducted a survey to gain feedback from the users. A total of 28 senior undergraduate and graduate Computer Science students (with internship and real-world software development experiences) participated in this survey where they tested SoCeR by uploading Python source code from their own software projects. They also searched for related functions at different stages of uploading their code using natural languages and specific requirements from their own projects. The three questions used for the survey cover the following aspects: 1. *If they had any prior experience with any source code recommendation system*, 2. *Did they get the expected outputs from SoCeR based on their queries*, and 3. *If they would be able to use any of these search results (i.e. source code) on a different project*. The participants gave their feedback on a scale of 0 to 5 where 0 represents strongly disagree and 5 represents strongly agree for the last two questions and yes/no for the first question. After compiling the survey results, we found that majority of the participants (23 out 28) did not use a code recommendation system before. For the second question, 53.6% of the participants gave 4 out of 5 and 25% participants gave 5 out of 5. For the third question, 35.7% participants gave 4 out of 5 and 32.1% participants gave 5 out of 5. Rest of the participants responded to somewhat agree, i.e., a score of 3, for the last two questions. The survey result demonstrates the potential of SoCeR to search private code repositories using natural language query.

TABLE VI. SIMILARITY SCORE FOR COMBINATIONS OF PARAMETERS

DS = Docstring; FN = Function Name; CT = Comment; VAR = Variable Name	**Score of first search result**
Sample Query-1: "unified hand gesture detection"	
Single - DS/FN/CT/VAR	.245/.056/.267/.091
Double - CT+FN/CT+DS/CT+VAR	.263/.264/.260
Triple - CT+DS+VAR/CT+DS+FN	.352/.350
All	.360
Sample Query-2: "send email with attachment"	
Single - DS/FN/CT/VAR	.234/.249/.314/.114
Double- CT+FN/CT+DS/CT+VAR	.431/.417/.344
Triple - CT+DS+VAR/CT+DS+FN	.438/.514
All	.529
Sample Query-3: "find shortest path in graph"	
Single - DS/FN/CT/VAR	.132/.173/.174/.188
Double- CT+FN/CT+DS/CT+VAR	.291/.243/.230
Triple - CT+DS+VAR/CT+DS+FN	.278/.279
All	.338

TABLE VII. PRECISION OF SAMPLE QUERIES

Sample Query	Returned Results	Related Results	Precision
"how to implement sort"	14	10	71.43
"database operation (without getter,setter)"	15	13	86.67
"detect person activity"	20	14	70.00

VI. CONCLUSION

In this paper, we presented a technique for searching Python code in private repositories using natural language. The main contributions of this paper include a code descriptor to represent code summary and a code recommendation technique which takes natural language as input and returns relevant source code from local repositories. We also incorporated a query suggestion feature which suggests related queries based on initial textual query and relevant code descriptors. Instead of searching the source code directly, our novel approach finds semantic similarity between the user query and the descriptor of the source code to return relevant results with high similarity scores. Since our goal is to promote reliable code reuse based on a verified code base, SoCeR does not rely on crowdsourced repositories as observed in the existing code recommendation tools and summarization techniques. SoCeR users do not need to have knowledge of the software structure or workflow, instead they can use software requirements as query to get relevant implementation or code snippets even before the actual coding phase starts. SoCeR returns acceptable results even if the source code lack some parameters. Additionally, it handles nested and recursive functions, functions with multiple branches, and different versions of the source code. Last but not the least, SoCeR does not recommend source code that has syntax errors. In the future, our goal is to extend the functionalities of SoCeR for other structured programming languages.

REFERENCES

[1] J. Stylos and B. A. Myers, "Mica: A Web-Search Tool for Finding API Components and Examples," in Visual Languages and Human-Centric Computing, 2006, pp. 195–202.

[2] 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 – Vol. 1, New York, NY, USA, 2010, pp. 475–484.

[3] R. Holmes and G. C. Murphy, "Using structural context to recommend source code examples," in Proceedings. 27th International Conference on Software Engineering (ICSE), 2005, pp. 117–125.

[4] S. Thummalapenta and T. Xie, "Parseweb: A Programmer Assistant for Reusing Open Source Code on the Web," in Proceedings of the Twenty-second IEEE/ACM International Conference on Automated Software Engineering, New York, NY, USA, 2007, pp. 204–213.

[5] The Koders source code search engine, 2005. http://www.koders.com

[6] The Krugle source code search engine. http://www.krugle.com

[7] M. Raghothaman, Y. Wei, and Y. Hamadi, "SWIM: Synthesizing What I Mean: Code Search and Idiomatic Snippet Synthesis," in Proceedings of the 38th Intl. Conference on Software Engineering, 2016, pp. 357–367.

[8] E. Linstead, S. Bajracharya, T. Ngo, P. Rigor, C. Lopes, and P. Baldi, "Sourcerer: mining and searching internet-scale software repositories," *Data Min Knowl Disc*, vol. 18, no. 2, Apr. 2009, pp. 300–336.

[9] O. A. L. Lemos et al., "CodeGenie: Using Test-cases to Search and Reuse Source Code," in Proceedings of the 22nd Intl. Conference on Automated Software Engineering (ASE), 2007, pp. 525–526.

[10] A. Zagalsky, O. Barzilay, and A. Yehudai, "Example Overflow: Using social media for code recommendation," in International Workshop on Recommendation Systems for Software Engineering, 2012, pp. 38–42.

[11] M. McCandless, E. Hatcher, and O. Gospodnetic, Lucene in Action, Second Edition: Covers Apache Lucene 3.0. Greenwich, CT, USA: Manning Publications Co., 2010.

[12] H. C. Wu, R. W. P. Luk, K. F. Wong, and K. L. Kwok, "Interpreting TF-IDF Term Weights As Making Relevance Decisions," in ACM Trans. Inf. Syst., vol. 26, no. 3, 2008, pp. 13:1–13:37.

[13] Y. Oda et al., "Learning to Generate Pseudo-Code from Source Code Using Statistical Machine Translation (T)," in 30th IEEE/ACM Intl. Conference on Automated Software Engineering, 2015, pp. 574–584.

[14] S. P. Reiss, "Semantics-based code search," in Proceedings of 31st Intl. Conference on Software Engineering, USA, 2009, pp. 243–253.

[15] Raphael Sirres, Tegawendé F. Bissyandé, Dongsun Kim, David Lo, Jacques Klein, and Yves Le Traon, "Augmenting and Structuring User Queries to Support Efficient Free-Form Code Search", Empirical Software Engineering Journal, Vol. 23, No. 5, 2018, pp. 2622–2654.

[16] M. M. Rahman and C. Roy, "Effective Reformulation of Query for Code Search Using Crowdsourced Knowledge and Extra-Large Data Analytics," 2018 IEEE International Conference on Software Maintenance and Evolution (ICSME), Madrid, 2018, pp. 473-484.

[17] Ragkhitwetsagul, C., Krinke, J. Siamese: scalable and incremental code clone search via multiple code representations. Empirical Software Engineering Journal, Vol. 24, 2019, pp. 2236–2284.

[18] Paul W. McBurney and Collin McMillan, "Automatic documentation generation via source code summarization of method context," in Proceedings of the International Conference on Program Comprehension (ICPC), 2014, pp. 279-290.

[19] G. Sridhara, E. Hill, D. Muppaneni, L. Pollock, and K. Vijay-Shanker, "Towards automatically generating summary comments for Java methods," in Proceedings of the IEEE/ACM international conference on Automated software engineering (ASE), 2010, pp. 43-52.

[20] L. Mou L, R. Men, G. Li, L. Zhang, Z. Jin, "On end-to-end program generation from user intention by deep neural networks," arXiv preprint arXiv:1510.07211. 2015.

[21] https://pypi.org/project/pycodestyle/

[22] https://pypi.org/project/autopep8/0.8/

[23] https://www.python.org/dev/peps/pep-0008/

[24] https://docs.python.org/3/library/ast.html

[25] Christiane Fellbaum (1998, ed.) WordNet: An Electronic Lexical Database. Cambridge, MA: MIT Press.

[26] https://www.nltk.org/_modules/nltk/corpus/reader/wordnet.html

[27] https://scikit-learn.org/stable/modules/generated/sklearn.feature_extraction.text.TfidfVectorizer.html

[28] G.W. Furnas, T.K. Landauer, L.M. Gomez, L.M. and S.T. Dumais, "The vocabulary problem in human-system communication", in Communications of the ACM, 30(11), 1987, pp.964-971.

[29] D. Liu, A. Marcus, D. Poshyvanyk, and V. Rajlich, "Feature location via information retrieval based filtering of a single scenario execution trace", in Proceedings of the twenty-second IEEE/ACM international conference on Automated software engineering, 2007, pp. 234-243.

[30] 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 International Conference on Software Engineering, San Francisco, CA, USA, 2013, pp. 842–851.

[31] L. Nie, H. Jiang, Z. Ren, Z. Sun, and X. Li, "Query Expansion Based on Crowd Knowledge for Code Search," IEEE Trans. Serv. Comput., vol. 9, no. 5, Sep. 2016, pp. 771–783.

[32] M. M. Rahman, "Supporting code search with context-aware, analytics-driven, effective query reformulation," in Proceedings of the 41st International Conference on Software Engineering: Companion Proceedings, Montreal, Quebec, Canada, 2019, pp. 226–229.