A Similarity-Based Method Retrieval Technique to Improve Effectiveness in Code Search

Abdus Satter
Institute of Information Technology
University of Dhaka
Dhaka, 1000
bit0401@iit.du.ac.bd

Kazi Sakib
Institute of Information Technology
University of Dhaka
Dhaka, 1000
sakib@iit.du.ac.bd

ABSTRACT

The effectiveness of a code search engine is reduced if semantically similar code fragments are not indexed under common and proper terms. In this paper, a technique named Feature-Wise Similar Method Finder (FWSMF) is proposed which checks functional similarity among codes by executing and matching outputs against the same set of inputs. It then determines appropriate index terms by finding keywords that are found in most of the code snippets. As a result, code fragments that contain different keywords but implement the same feature, can be retrieved all together. Experimental analysis shows that on an average, FWSMF produces 61% and 29% more precision than two existing techniques named Keyword Based Code Search (KBCS) and Interface Driven Code Search (IDCS) respectively. It also shows 34% and 55% more recall than KBCS and IDCS correspondingly. It retrieves self executable code snippets which can be easily pluggable in the intended development context and thus reduces time and effort while reusing code.

KEYWORDS

Code Search, Code Reuse, Self-Executable Code

1 INTRODUCTION

The effectiveness of a Code Search Engine (CSE) depends on the number of retrieved code fragments that are relevant to the user need. The reason is that developers need existing code snippets to understand the implementation of a particular feature, to know the usage of an Application Programming Interface, or to reuse these with some adaptations in the development context [5]. A CSE should retrieve as many relevant code fragments as possible so that developers can choose the one that best satisfies their needs. Usually, traditional Information Retrieval (IR) centric approaches are employed by the CSE where keywords found in the code snippets (i.e., method name, variable name, etc.) used to construct the index [15]. In these approaches, a collection of code fragments that perform the same feature but do not contain the same keywords, will be indexed against different terms. When a query term matches one of these index terms, only the corresponding code fragment will be retrieved instead of all these fragments. Many relevant code fragments cannot be retrieved due to this keyword matching policy and thus the effectiveness of the code search engine is reduced.

In order to improve the effectiveness of a code search engine, feature-wise similar code fragments should be indexed under common and proper terms. This will retrieve all the relevant code snippets together that contain different keywords. However, it is challenging to determine feature-wise similarity among the syntactically different code fragments because it requires determining the implemented feature of these fragments [11]. Another challenge is to select proper term that represents the intent of a code fragment properly. For example, three methods named “bubble”, “bbl”, and “bubbleSort” sort an array of elements. The term “bubble” is ambiguous, “bbl” is not consistent with human perceivable language dictionary. However, the term ”bubble sort” expresses the implemented feature properly and it should be used to index all these methods. Selecting such proper term automatically for indexing is also a research challenge.

Several techniques have been found in the literature to improve the effectiveness in code search which can be classified into Keyword Based Code Search (KBCS), Interface Driven Code Search (IDCS), and Semantic based Code Search (SBCS). KBCS [1, 2, 10, 14, 15] considers the source code as plain text document and constructs index following IR centric approaches. IDCS [4, 16, 17] helps to refine the user query by explicitly telling the interface of required code snippets (such as signature of a method). SBCS [6, 7, 12, 13] runs user provided test cases on the codes retrieved by IDCS to find the semantically relevant code fragments. None of these techniques considers functional similarity among code fragments, and index term appropriateness when constructing index. Thus, these miss many semantically relevant codes that do not contain proper keywords.

2 PROPOSED TECHNIQUE

In this paper, a technique named Feature-Wise Similar Method Finder (FWSMF) is proposed to find the semantically similar but syntactically different methods. The technique comprises four steps which are described as follows.

Self-Executable Method Generation: The technique starts with parsing the source files to identify all the methods in a given code base. For each method (m), a ***Call Graph*** is generated to identify the methods invoked by m directly or indirectly. A ***Data Dependency Graph (G)*** is also constructed for m to find the fields that are declared outside the method body but used by it. Each node in G denotes a field or variable, and an edge $a \rightarrow b$ expresses that a depends on b. All the libraries are identified on which m is

Programming '17, Brussels, Belgium
 978-1-4503-4836-2/17/04…$15.00
DOI: http://dx.doi.org/10.1145/3079368.3079372

dependent for execution. Method, field, and data dependencies are accumulated for m to make it self-executable.

Method Signature Regeneration: Although the signature of a method expresses the input and output types of the method, this is not sufficient enough for several cases. A method may contain no parameter in its signature but may manipulate one or more variables and/or fields. Again, a method may contain return type *void* but its task is to change the value of a field. To explicitly know the input and output types, the signature of a self-executable method is regenerated by constructing ***Data Dependency Graph*** for the method. Nodes in the graph that express the fields and have in-degree zero are considered as parameters of the method. A complex data type is created to denote return type where nodes that have out-degree zero and represent fields, are added as members of the complex data type. Finally, if the method body contains *return* statement, it is replaced with the new complex data type otherwise added to the end of the method body to incorporate the changes.

Clustering Similar Methods: For a given set of self-executable methods (M) obtained following the previous steps, similarity is checked by running each method $m \in M$ and checking the output. Initially, a set of input values (I) is generated for $m \in M$ and corresponding set of output values (O) is obtained through executing m. For each $m' \in M$ and $m \neq m'$, m' is said to be functionally similar if its output set O' for I is the same to O. Accordingly, methods are clustered based their feature-wise similarities where each cluster contains the methods that perform the same task. That is, a cluster $C \subset M$ and $\forall x, y : C \cdot x$ and y are functionally similar. There may have different techniques to implement the same feature, for example, sorting can be implemented following bubble sort, merge sort, and etc. All the sorting techniques will be in the same cluster as these are functionally similar but implementation-wise different. It would not be good to retrieve codes implementing bubble sort when a user looks for merge sort. So, each cluster C is further decomposed into a set of clusters (R) based on time and memory space complexities. That is, $R \subseteq C$ and $\forall r : R \cdot (\forall p, q : r \cdot O(p) = O(q)$ and p, q are functionally similar).

Index Construction and Query Formulation: After obtaining all the clusters (T) from the previous step, keywords are extracted from the method name which are further tokenized and stemmed to generate terms. For each cluster, suitable terms are selected for indexing which appear in most of the methods of that cluster. A boolean query is expanded through WordNet to solve vocabulary mismatch problem. Finally, the expanded query is matched with the index to retrieve feature-wise similar codes.

3 EXPERIMENTAL SETUP AND RESULT ANALYSIS

To conduct experimental analysis, FWSMF was implemented in Java. 8 different features were selected from existing works [8–10, 12] (as shown in Table 1) and 25 masters students were employed to implement each of these features. Thus, the experimental code base contains 25 code snippets per feature and 200 code snippets in total. FWSMF was run on these codes to check the clustering purity. It constructed 8 clusters accurately where each cluster contains all the 25 implementations of the respective feature.

Table 1: Selected Functionalities with Number of Queries

Functionality	# queries	Functionality	# queries
decoding String	10	rotating array	15
encrypting password	20	resizing image	7
decoding a URL	16	scaling Image	21
generating MD5 hash	20	encoding string	11

Table 2: Result Analysis in Percentage

	KBCS	IDCS	FWSMF
Precision	35	67	96
Recall	59	38	93
# Retrieved Self-Executable Codes	28	75	100

Three open source projects (EGit[1], JGit[2], JUnit[3]) were added to the experimental code base to determine the precision and recall of FWSMF in identifying relevant codes written by the subjects. Two existing techniques KBCS and IDCS were also used for comparative result analysis. One of the common difficulties in reusing existing code is to make the code executable in the development context through resolving dependencies, and it induces significant amount of development time and cost. Developers should be provided more self-executable relevant codes so that they can use these easily without thinking dependencies required for execution. So, number of retrieved self executable codes is also considered here.

Subjects were asked to generate queries for each of the experimented features and evaluate the results in terms of relevance and self-executability. There were 120 user queries in total as shown in Table 1 and a summary of the results is shown in Table 2. On an average, FWSMF shows 34% and 55% more recall than KBCS and IDCS respectively. Besides, precision is improved by 61% and 29% more by FWSMF in comparison with KBCS and IDCS respectively. The reason for such results is that both KBCS and IDCS do not consider feature-wise similarity among code fragments during index construction and many relevant codes cannot be obtained due to indexing against inappropriate terms. KBCS and IDCS retrieves 28% and 61% self-executable code fragments correspondingly. However, All the code fragments retrieved by FWSMF are self-executable due to resolving function, data, and library dependencies.

4 CONCLUSION

This paper presents a technique named FWSMF which improves effectiveness in code search by indexing functionally similar codes under proper terms, and delivering self-executable codes to reduce development time and cost. According to the result analysis, it outperforms existing techniques - KBCS and IDCS in terms of precision, recall, and retrieved code quality. Although existing code clone detection approaches can improve the effectiveness, these provide false positive results if certain parameters' values are not set properly [3]. FWSMF checks dynamic behavior through executing source codes to ensure feature-wise similarity among these codes.

[1] https://www.github.com/eclipse/egit
[2] https://www.github.com/eclipse/jgit
[3] https://www.github.com/junit-team/junit4

REFERENCES

[1] 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*. ACM, 681–682.

[2] Andrew Begel. 2007. Codifier: a programmer-centric search user interface. In *Proceedings of the workshop on human-computer interaction and information retrieval*. 23–24.

[3] Yingnong Dang, Dongmei Zhang, Song Ge, Chengyun Chu, Yingjun Qiu, and Tao Xie. 2012. XIAO: tuning code clones at hands of engineers in practice. In *Proceedings of the 28th Annual Computer Security Applications Conference*. ACM, 369–378.

[4] Reid Holmes, Robert J Walker, and Gail C Murphy. 2005. Strathcona example recommendation tool. In *ACM SIGSOFT Software Engineering Notes*, Vol. 30. ACM, 237–240.

[5] Oliver Hummel, Werner Janjic, and Colin Atkinson. 2008. Code conjurer: Pulling reusable software out of thin air. *IEEE software* 25, 5 (2008), 45–52.

[6] Werner Janjic and Colin Atkinson. 2012. Leveraging software search and reuse with automated software adaptation. In *Search-Driven Development-Users, Infrastructure, Tools and Evaluation (SUITE), 2012 ICSE Workshop on*. IEEE, 23–26.

[7] Otávio Augusto Lazzarini Lemos, Sushil Krishna Bajracharya, and Joel Ossher. 2007. CodeGenie:: a tool for test-driven source code search. In *Companion to the 22nd ACM SIGPLAN conference on Object-oriented programming systems and applications companion*. ACM, 917–918.

[8] Otávio AL Lemos, Adriano C de Paula, Felipe C Zanichelli, and Cristina V Lopes. 2014. Thesaurus-based automatic query expansion for interface-driven code search. In *Proceedings of the 11th Working Conference on Mining Software Repositories*. ACM, 212–221.

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

[10] Otávio Augusto Lazzarini Lemos, Adriano Carvalho de Paula, Hitesh Sajnani, and Cristina V Lopes. 2015. Can the use of types and query expansion help improve large-scale code search?. In *15th International Working Conference on Source Code Analysis and Manipulation (SCAM)*. IEEE, 41–50.

[11] Erik Linstead, Paul Rigor, Sushil Bajracharya, Cristina Lopes, and Pierre Baldi. 2007. Mining concepts from code with probabilistic topic models. In *Proceedings of the twenty-second IEEE/ACM international conference on Automated software engineering*. ACM, 461–464.

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

[13] Naiyana Sahavechaphan and Kajal Claypool. 2006. XSnippet: mining for sample code. *ACM Sigplan Notices* 41, 10 (2006), 413–430.

[14] Susan Elliott Sim and Rosalva E Gallardo-Valencia. 2013. *Finding source code on the web for remix and reuse*. Springer.

[15] Renuka Sindhgatta. 2006. Using an information retrieval system to retrieve source code samples. In *Proceedings of the 28th international conference on Software engineering*. ACM, 905–908.

[16] Suresh Thummalapenta and Tao Xie. 2007. Parseweb: a programmer assistant for reusing open source code on the web. In *Proceedings of the 22nd IEEE/ACM international conference on Automated software engineering*. ACM, 204–213.

[17] Amy Moormann Zaremski and Jeannette M Wing. 1995. Signature matching: a tool for using software libraries. *ACM Transactions on Software Engineering and Methodology (TOSEM)* 4, 2 (1995), 146–170.