Improving Recall in Code Search by Indexing Similar Codes under Proper Terms

Abdus Satter* and Kazi Sakib†
Institute of Information Technology, University of Dhaka, Dhaka, Bangladesh
Email: *bit0401@iit.du.ac.bd, †sakib@iit.du.ac.bd

***Abstract*—The recall of a code search engine is reduced, if feature-wise similar code fragments are not indexed under common terms. In this paper, a technique named Similarity Based Method Finder (SBMF) is proposed to alleviate this problem. The technique extracts all the methods from a source code corpus and converts these into reusable methods (i.e., program slice) through resolving data dependency. Later, it finds similar methods by checking signature (i.e., input and output types) and executing methods for a randomly generated set of input values. Methods are considered as feature-wise similar if these produce the same output set. In order to index these methods against common and proper terms, SBMF selects the terms that are found in most of the methods. Finally, query expansion is performed before searching the index to solve the vocabulary mismatch problem. In order to evaluate SBMF, fifty open source projects implementing nine different functionalities or features were used. The results were compared with two types of techniques - Keyword Based Code Search (KBCS) and Interface Driven Code Search (IDCS). On an average, SBMF retrieves 38% and 58% more relevant methods than KBCS and IDCS, respectively. Moreover, it is successful for all the features by retrieving at least one relevant method representing each feature whereas IDCS and KBCS are successful for 3 and 7 features out of 9 respectively.**

***Index Terms*—code search, code reuse, method search**

I. Introduction

The recall of a code search engine, indicated by the number of relevant codes that is retrieved from the code repository, usually depends on the indexing mechanism and query formulation techniques. Proper indexing and query understanding help to retrieve relevant code snippets that satisfy user needs [1]. Most of the code search engines employ Information Retrieval (IR) centric approaches for indexing source code [2]. The working principle behind these approaches is to construct a term-based index, by extracting keywords from source codes. A common problem of these approaches is that a pair of codes - having same functionality, but written using different keywords are indexed against different terms. A traditional code search engine misses some important code fragments, because of this keyword matching policy. It results in a low recall code search engine with poor performance on benchmark datasets [3].

To improve recall of a code search engine, similar code fragments should be indexed under the same terms. However, it is challenging to automatically and efficiently determine that two code fragments are identical or similar [4]. Although identical code fragments can be detected through keywords matching [5], detecting feature wise similar code blocks is difficult. The reason is that automatically perceiving the intent of a code block is still a research challenge [6]. Another challenge is to select proper terms that best represent similar code fragments. For example, assume that there are two methods that perform bubble sort - "x" and "sort". Here, between two terms, "sort" is semantically more relevant name than "x". It is a challenging task to automatically determine that "sort" is the better keyword to represent these methods. Again, a code fragment may contain terms, which are not useful to express its intent (i.e., implemented feature) properly. Indexing based on these keywords reduces matching probability between user query and these keywords. It happens because, user query defines functionality but the extracted keywords do not express the feature properly. So, instead of using these keywords, more meaningful terms need to be selected that best match the query.

Researchers have proposed various techniques to improve the performance of code search engines where recall is considered as one of the performance indicators. These techniques can be broadly classified into four types like Keyword Based Code Search (KBCS), Interface Driven Code Search (IDCS), Test Driven Code Search (TDCS), and Semantic Based Code Search (SBCS). In KBCS [2], [7], [8], [9], [10], source codes are indexed based on the terms generated from the code and searching is performed on the index. As this approach does not consider similarity between source codes having different keywords, it cannot retrieve more relevant codes. In order to define required component interface as query, and find relevant components, IDCS [11], [12], [13] was proposed. It is possible to have two or more code fragments that contain different interfaces but perform the same task. IDCS considers that these code fragments are different due to having different interfaces. Thus, it does not retrieve these all together. To automatically find and adapt reusable components, TDCS [14], [15] and SBCS [16], [17] were proposed. These are effective in terms of precision as test cases are employed on the retrieved codes. In these approaches, most of the test cases fail not only for functional requirements mismatch but also for syntactic mismatch of the interface definition [15]. For this reason, semantically relevant code fragments cannot be retrieved and the recall is decreased.

In this paper a technique named Similarity Based Method Finder (SBMF) is proposed to retrieve more relevant methods from code base. The technique first parses all the methods from the source code to construct a repository of methods. It generates data dependency graph for each method and

converts the method into reusable method (i.e., program slice) through resolving data dependency, and redefining parameters and return type. Later, all the methods are clustered into a number of clusters where methods in the same cluster perform the same task. To detect feature-wise similarity among a set of methods' signatures (i.e., parameters and return types) of these methods are checked. Methods having the same signature are then executed against a set of randomly generated input values. Among these methods, those which produce the same output are considered as feature-wise similar and a cluster is constructed to store these methods. To identify proper terms for a cluster, keywords are obtained from the methods in the cluster and method frequency is calculated for each term. Such terms are considered as representative terms if these are found in most of the methods of the cluster. All the methods of the cluster are then indexed against the terms so that these are retrieved all together if a query term matches one of these methods. At last, user query is expanded by adding synonyms of each query term to increase the matching probability between the query terms and index terms [7].

In order to evaluate the proposed technique, a tool was developed. Two types of code search techniques, KBCS and IDCS, were compared with SBMF to show its efficiency. An existing system named Sourcerer [8] was used for the implementation of KBCS and IDCS. However, SBCS and TDCS were not considered for comparison, because these were proposed to improve precision rather than recall. For comparative result analysis, three metrics were used which are *recall*, *number of methods retrieved* and *feature successfulness*. Here, *feature successfulness* determines whether at least one relevant method is retrieved or not against user queries provided for a feature. In the context of this paper, *A feature can be considered as a requirement given to a developer to implement*. 50 open source projects were selected to carry out the experiment. The result analysis shows that on an average SBMF increases recall by 38% and 58% more than KBCS and IDCS, respectively against 170 queries. Besides, SBMF is successful for all the features whereas KBCS and IDCS are successful for 7 and 3 features out of 9 respectively.

II. Related Works

Reusing existing code fragments reduces development time and effort [18]. For this reason, searching for reusable code snippets has become a common task among the developers during software development [19]. Various techniques have been proposed in the literature to improve the performance of code search engine in terms of recall, precision, query successfulness, etc. These techniques can be broadly classified into four categories which are Keyword Based Code Search (KBCS), Interface Driven Code Search (IDCS), Semantic Based Code Search (SBCS), and Test Driven Code Search (TDCS). Significant works related to each category are discussed in the following subsections.

A. Keyword based Code Search (KBCS)

In KBCS, source code is considered as plain text document where traditional IR centric approaches are employed to index the code and query over the index [20]. Besides, other metadata such as comments, file name, commit message, etc. are used to retrieve relevant code fragments from a repository of source codes. One of the techniques related to KBCS is JSearch which indexes source code against the keywords extracted from the code [2]. However, it cannot retrieve all the code snippets that implement the same feature but contain different keywords. This is because, it does not check feature-wise similarity to detect common terms for these fragments.

Several techniques like Sourcerer [8], Codifier [9], krugle [10], etc. were proposed to provide infrastructure for large scale code search. These techniques use both structural and semantic information of source code to construct index. Structural information comprises language, source file, related documents, classes, methods, dependencies, and so on. Semantic information of a program is gathered by generating terms from method name, class name, field name, comments, etc. Although these techniques adopt both types of information to fetch more relevant code fragments, these cannot retrieve feature-wise similar code blocks simultaneously. The reason is that all these information are stored following IR based indexing mechanism, and no checking is performed to index similar code snippets under common proper terms.

B. Semantic Based Code Search (SBCS)

As open source codes are increasing day by day, it is thought that a significant amount of code that is written today, has already been available in the internet. However, reusing these existing codes often does not directly meet user needs or requires modifications. In order to find existing codes that support user requirements, a technique in form of SBCS was proposed by Steven [16]. It takes keywords that represent user requirements, and retrieves relevant code fragments containing these keywords. Later, it runs user provided test cases on the fetched code snippets and passed codes are delivered as final search result. It performs well in terms of precision but recall is reduced since proper terms are not determined while indexing feature-wise similar codes. So, some semantically similar code fragments cannot be fetched due to indexing these under inappropriate terms.

Sometimes, developers need to convert one type of object to another. To get example code implementing such conversion, Niyana proposed a technique named XSnippet [17]. It creates graph from source code by adopting code mining algorithm. The graph represents data flow within the corresponding source code. Moreover, user query is defined by providing input type and output type. For a user query, all the generated graphs are searched to find those code fragments that convert the input type into the output type. In this technique. developers need to provide exact input type and output type for getting example code blocks. Otherwise, it cannot retrieve code fragments that may satisfy user needs. However, according to the searching behavior, developers are

more interested in using keywords rather than concrete data types to define their query [21].

C. Test Driven Code Search (TDCS)

TDCS is a special type of SBCS where test cases are used to obtain program semantics. Lemos et al. proposed a TDCS technique named CodeGenie to support method level searching [14]. The technique takes method signature as query from the test cases written by developers. It uses Sourcerer infrastructure to retrieve relevant functions against the query. Next, all the test cases are executed for each retrieved method. Resultant methods are ranked based on the number of test cases successfully passed. Although the technique increases precision, it produces low recall. The reason is that it performs keyword matching to fetch methods from index without justifying the appropriateness of the keywords.

Usually retrieved methods may not pass corresponding test cases due to different order of the parameters, return type or parameter type. To resolve these issues, Janjic et al. proposed a technique that refactors the code to adapt with the program context [15]. It applies every possible adaptations like reordering parameters, using super type or sub class type of a given return or parameter type, converting primitive type to reference type, etc. Thus, it improves TDCS by finding more relevant methods. However, it produces low recall because it does not index similar methods under common terms.

D. Inreface Driven Code Search (IDCS)

IDCS helps the developers to define their queries in a more structured form rather than just a set of keywords joined by boolean expression. Signature matching was the first proposed IDCS technique to find relevant functions within a software library [13]. The approach crawls all the methods in the library, and uses signature of each method for indexing. Other code search techniques such as Sourcecer, ParseWeb, and Strathcona also support IDCS to improve the performance in code search [7]. Although IDCS assists to formulate user query, it does not select appropriate terms during indexing similar codes that perform the same functionality. Thus, functionally related code fragments will not be retrieved all together since these are indexed against inappropriate terms.

In order to find reusable code fragments, four types of techniques have been proposed in the literature which are KBCS, IDCS, TDCS, and SBCS. All these techniques extract keywords from source code to generate terms, and index corresponding code against the terms. However, none of the techniques checks the appropriateness of the terms with respect to implemented feature. As a result, the number of relevant codes retrieved is reduced due to indexing against improper term. Moreover, if two or more code snippets implement similar feature but contain different terms, existing techniques cannot retrieve all these code fragments simultaneously. The reason is that these are indexed against different terms. So, to improve recall in code search, feature-wise similar codes should be indexed under common appropriate terms.

III. Proposed Technique

In this paper, a technique named Similarity based Method Finder (SBMF) has been proposed to improve recall in code search. The technique comprises several steps such as *Reusable Method Generation*, *Clustering Similar Methods*, *Proper Term Selection*, *Handling Methods Having API/Library Function call*, *Index Construction*, and *Query Expansion*. Each of the steps is discussed as follows.

A. Reusable Method Generation

In this step, the proposed technique first parses the source code to identify all the methods in the code. For each method, it checks whether the body of that method contains any API/-function call statement or not. If no such statement is found, the technique takes the method to convert it into reusable method (i.e., program slice that can execute independently without having any dependency on other methods). Later, a data dependency graph is constructed for the corresponding method to determine its input and output types. Although the signature of the method expresses the input and output types, this is not sufficient enough to convert into reusable function for several scenarios. For example, a method may have return type void but it may manipulate one or more variables that are declared outside the body of the method. A method may not have any parameter (i.e., void) but use variables that are defined outside the body of the method. Again, the signature of a method may explicitly state the input and output types but some variables may be used or manipulated by it and these are declared outside the method body. Considering all of these scenarios, the technique generates data dependency graph to redefine the signature and convert into reusable method. Each node in the graph denotes the variable and an edge from a to b ($a \rightarrow b$) denotes variable a depends on variable b. After constructing the graph, nodes that have in degree zero and variables denoted by these nodes are declared outside the method body, are considered as input parameters. Besides, nodes that have out degree zero are considered as output variables of the method. If multiple output nodes are found, a complex data type is created where each field of the type denotes each node. The reason is that a method return type can be a single data type - either primitive or complex data type. The technique uses the variables found in the nodes containing in degree zero to generate parameters of the method. If a single node is found which out degree is zero, the type of the variable denoted by the node is used as return type of the method. Otherwise, generated composite data type as discussed earlier is used. The signature of the method is redefined by combining the return type, method name, and parameters. It is possible to have one or more variables that are declared outside the method body. In the data dependency graph, nodes representing these variables may have at least one in degree and one out degree. In this case, the technique parses the source code and checks the declaration statements of the variables to determine the types of the variables. Using this information, it adds declaration statement for each of the variables at the beginning of the function body. Thus, the

technique makes the method self-executable without having any external data dependency.

B. Clustering Similar Methods

To improve code search, it is required to check the similarity among methods found in the code base. Two or more methods may perform the same task in different ways. So, feature-wise similar methods needs to be detected to retrieve the similar methods all together. In Algorithm 1, the procedure named $ClusterSimilarMethods$ takes a list of reusable methods (M) as input which is constructed following the previous step. A variable C is declared to store different clusters of similar methods where each cluster contains the methods that perform the same functionality (Algorithm 1 Line 2). A *for* loop is declared that iterates on M to construct cluster of similar methods. The procedure $IsInAnyCluster$ is invoked to check whether each method m (belongs to M) is added to any cluster or not previously (Algorithm 1 Lines 4-5). If m does not belong to any cluster, a variable cl is declared to contain all the methods similar to m. A set of input data is generated based on the type of parameters found in the signature of m and corresponding output is generated by executing m (Algorithm 1 Lines 9-10). Here $inputset$ and $outputset$ determine the intent of m. Another *for* loop is declared to identify other methods that are similar to m. In each iteration, the signature of each method m' (in M) is matched with the signature of m to check whether the input data set can be fed into the method and return type is identical to m (Algorithm 1 Line 15). If the signatures of both methods are identical, the method m' is executed for $inputset$ and generated output is stored to $outputset'$. If $outputset$ and $outputset'$ are found the same, m' is considered similar to m as both methods produce same output for the same input data set (Algorithm 1 Lines 17-19). m' is then added to cl to store all the methods similar to m. At last, cl is inserted to the list of all identified clusters (C).

Algorithm 1 Cluster Similar Methods

Require: A list of methods (M) for which search index will be constructed

1: **procedure** CLUSTERSIMILARMETHODS(M)
2: $C = \emptyset$;
3: **for** each $m \in M$ **do**
4: **if** $IsInAnyCluster(m, C) == true$ **then**
5: continue
6: **end if**
7: $cl = \emptyset$
8: $cl.add(m)$
9: $inputset$ = generate a set of input data randomly for m
10: $outputset$ = execute m and generate corresponding output for $inputset$
11: **for** each $m' \in M$ **do**
12: **if** $IsInAnyCluster(m', C) == true$ **then**
13: continue
14: **end if**
15: **if** $m'.paramtersTypes == m.parametersTypes$ & $m'.returnType = m.returnType$ **then**
16: $outputset'$ = execute m' and generate corresponding output for $inputset$
17: **if** $outputset == outputset'$ **then**
18: $cl.add(m')$
19: **end if**
20: **end if**
21: **end for**
22: $C.add(cl)$
23: **end for**
24: **end procedure**
25: **procedure** ISINANYCLUSTER(m, C)
26: $found = false$
27: **for** $c \in C$ **do**
28: **if** $m \in c$ **then**
29: $found = true$
30: break;
31: **end if**
32: **end for**
33: **return** $found$
34: **end procedure**

C. Proper Term Selection

In order to retrieve more relevant methods, it is required to identify proper terms for each method before indexing. When two or more methods have different names or signatures, but implement the same functionality, these methods should be indexed under common appropriate terms. As a result, all these methods will be obtained against user query. So, after getting all the clusters from the previous step, representative terms are selected for each cluster. For a cluster, terms are obtained from the methods found in the cluster through extracting, tokenizing, and stemming keywords found in the methods. Terms that are found in most of the methods are considered as final representative terms for each of these methods.

D. Handling Methods Having API/Library Function call

As developers also search for example code to understand the usage of an API, in this step, methods that have API call statements are gathered. For each identified method, terms are generated from API call statements to index against the terms. As a result, if a query term does not match with the signature of a method but matches with the API invocation statements, the method is retrieved as API usage example code.

E. Index Construction and Query Expansion

After generating appropriate terms for each method and merging similar ones, an index is built for searching desired methods. A posting list is created to construct index, which maps terms with corresponding methods. Later, user query is expanded to retrieve more relevant methods against the query.

Two procedures named $ConstructIndex$ and $Query$, are presented in Algorithm 2 to build index of methods obtained from the previous steps, and refine user query, respectively. To

construct the index, an empty posting list is declared, which maps each term to corresponding methods (Algorithm 2 Lines 2). A nested *for* loop is defined, where the outer loop iterates on a list of methods (M) given as input to the procedure (Algorithm 2 Lines 3-4). The inner loop iterates to get all the terms of each method in M. In addition, each term is checked whether the posting list contains it or not to add a new term in the list (Algorithm 2 Lines 5-7). Next, the method is added to the posting list against the term so that, when a query term will match with the term, the method will be retrieved (Algorithm 2 Lines 8). After adding all the methods, the list is returned by the procedure (Algorithm 2 Lines 11).

In procedure $Query$, a boolean query is given as an argument, from which terms are separated and stored in a variable named $queryTerms$ to expand the query (Algorithm 2, Lines 13-14). A nested *for* loop is defined, where the outer loop iterates on these terms (Algorithm 2, Lines 15-17). In each iteration, a temporary variable ($expandedTerm$) initialized with the corresponding term, is used to store synonyms of the term. To expand each term, synonyms are appended to $expandedTerm$ in the inner loop (Algorithm 2, Lines 17-19). Later, each term in $queryTerms$ is replaced with corresponding $expandedTerm$ for the expansion of the query (Algorithm 2, Lines 20). As a result of the expansion, the probability of matching a query string with the terms defined in the index increases. Finally, the query is executed in the index to retrieve intended methods, which are returned by the procedure (Algorithm 2, Lines 22-23).

Algorithm 2 Index Construction and Query Expansion

Require: A list of methods (M) containing signature, body and terms of each method

```
1: procedure CONSTRUCTINDEX(M)
2:     Map < String, List < Method >> postingList
3:     for each m ∈ M do
4:         for each t ∈ m.terms do
5:             if !postingList.keys.contains(t) then
6:                 postingList.keys.add(t)
7:             end if
8:             postingList[t].add(m)
9:         end for
10:    end for
11:    return postingList
12: end procedure
13: procedure QUERY(booleanQueryStr)
14:    queryTerms=get all terms from booleanQueryStr
15:    for each qt ∈ queryTerms do
16:        expandedTerm = qt
17:        for each syn ∈ synonyms of qt do
18:            expandedTerms+ =" OR "+syn
19:        end for
20:        queryTerms.replace(qt, expandedTerm)
21:    end for
22:    methods = obtain method from the index satisfying
       queryTerms
23:    return methods
24: end procedure
```

IV. Implementation and Result Analysis

In order to perform comparative result analysis, the proposed technique (SBMF) was implemented in form of a software tool. 50 open source projects were selected as data sources for the experimental analysis. To evaluate the proposed technique, 170 queries representing 9 different features were executed by the tool. For comparative analysis, same queries were also run on Sourcerer that supports KBCS and IDCS.

A. Environmental Setup

This section outlines the softwares and frameworks required for the experimental analysis. SBMF was implemented using C# programming language. Moreover, some other tools were also used, which are addressed as follows:

- JavaParser: An open source library used to parse Java source code (https://github.com/javaparser)
- Apache Lucene: A popular search engine infrastructure used to index java methods and query over the index (https://lucene.apache.org/)
- Luke: Open source lucene client used to execute query on the lucene index and visualize the search results (https://github.com/DmitryKey)

B. Dataset Selection

In order to perform experimental analysis, 50 open source projects from sourceforge (https://sourceforge.net/) were selected. Fraser and Arcuri showed that these projects are statistically sound and representatives of open source projects [22].

A set of features were selected from the existing works in code search [7], [16], [23], [24] as shown in Table I. On the other hand, to evaluate the proposed technique, a set of queries is selected from [7]. Here, each query is related to a particular functionality shown in Table I and all the queries are created randomly. 15 subjects were employed to identify relevant methods for the functionalities. Among 15 subjects, 5 of them were senior Java developers and rest 10 were masters student. The reason of choosing students in this study is that they can play important role in software engineering experiments [25]. All the experimental datasets are available in this link[1].

C. Comparative Result Analysis

For comparative result analysis, SBMF was run on the experimental datasets and the relevance of retrieved methods were checked for each user query. Moreover, Sourcerer which supports KBCS and IDCS, was also run on the same datasets and search results obtained by this were compared to SBMF. Three metrics were used to evaluate the performance of SBMF in comparison with KBCS and IDCS. These were recall, number of retrieved methods, and feature successfulness. Detailed

[1] http://tinyurl.com/zdqmoqz

TABLE I
SELECTED FUNCTIONALITIES WITH FREQUENCY

#	Functionality	# methods	# queries
1	decoding String	13	20
2	encrypting password	3	27
3	decoding a URL	3	21
4	generating MD5 hash	3	16
5	rotating array	2	25
6	resizing image	3	25
7	scaling Image	3	19
8	encoding string to html	2	6
9	joining string	47	36

result analysis with respect to each of the metrics is discussed as follows.

Recall Analysis: Recall is one of the most commonly used metrics to measure the performance of traditional IR system. As the intent of the paper is to improve recall in code search, it is considered as an important metric to evaluate the proposed technique. In this experiment, recall is defined as follows.

$$recall = \frac{\text{number of retrieved relevant methods}}{\text{number of relevant methods in the repository}}$$

Fig. 1 depicts a comparative recall analysis among SBMF, KBCS, and IDCS where X axis denotes the feature no. as shown in TABLE I and Y axis represents the measured recall. For feature 1 (Decoding String), approximately 15% recall is shown in Fig. 1 for both KBCS and IDCS whereas 100% recall is found for SBMF. There are 13 methods in the repository that implement the feature. Among these, two methods are found which contain keywords *decode* and *string* in method name and parameter respectively. As a result, these methods are retrieved by both KBCS and IDCS. However, these techniques cannot retrieve other 11 methods because signatures of these methods do not contain any term related to *decode*. While analyzing the source code of these methods, it is seen that the bodies of these methods use third party APIs like *URLDecoder.decode(String, String), Hex.decode(String), Base64.decode(base64), etc.* to implement the feature. SMBF takes terms from API call statements and indexes against the terms to provide example codes regarding API usage. So, it retrieves all these 13 methods.

For feature 2 (Encrypting Password), IDCS cannot find any methods but 66.67% and 33.3% relevant methods are retrieved by SBMF and KBCS respectively as shown in Fig. 1. To get the methods that implement this feature, the following query is provided to IDCS.

name:(encrypt) AND return:(String) AND parameter:(String)

Although there is a single method found in the code base that has *encrypt* keyword in its name but does not have *String* in its parameter. So, IDCS cannot obtain this method but KBCS retrieves because query keyword matches with the method name. However, SBMF retrieves one more method having signature *crypt(String strpw,String strsalt)*. The reason is that *encrypt* and *crypt* both express the same intent as detected by the query expansion part of SBMF (Algorithm 2).

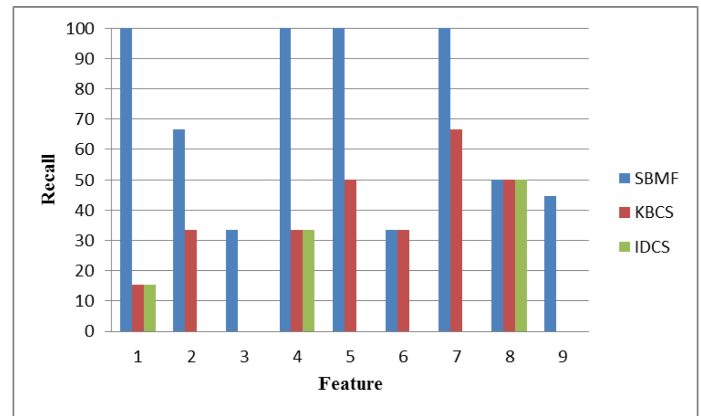

Fig. 1. Recall Analysis

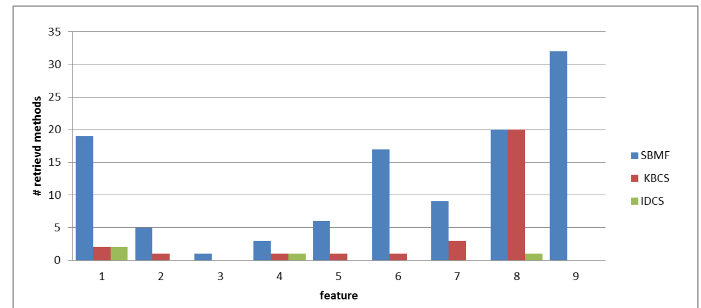

Fig. 2. Number of Retrieved Methods

There are 3 relevant methods in the experimental projects that implement feature no. 3 (Decoding a URL). According to Fig. 1 only a single method is retrieved by SBMF that produces recall 33.33%. On the contrary, KBCS and IDCS cannot retrieve any method related to the feature. This is because no method contains *decode* and *URL* simultaneously in the signature. Although one of these methods named *getPath* does not provide any semantic information representing the feature, it invokes a library method - *URLDecoder.decode(path, "UTF-8")* which implements the feature. SBMF considers the invocation statement for getting more relevant terms and thus, retrieves this method. Two other methods cannot be retrieved by SBMF due to finding no structural similarity among these and no keywords representing the feature.

According to Fig. 1, 100% recall is obtained for SBMF, and 33.33% for KBCS and IDCS individually with respect to feature no. 4 (Generating MD5 hash). It is clear that SBMF

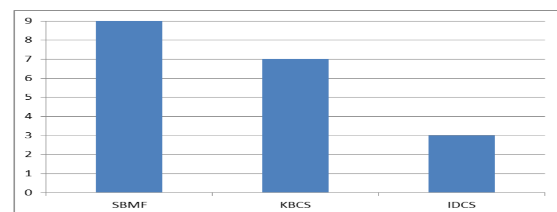

Fig. 3. Feature Successfulness Analysis

has higher recall than other two approaches. The reason is that most of the methods implementing this feature do not have proper names to represent their intent. There are 5 methods relevant to this feature and only one method is found having name consistent with the feature. KBCS and IDCS fail to retrieve all these methods because both techniques extract terms from individual method and do not consider appropriateness of the terms. However, SBMF finds that these methods are semantically similar. So, these methods are indexed under common terms. As a result, when user query matches with one of these methods, other three methods are also retrieved with this.

For feature no. 5, 6 and 7, IDCS cannot retrieve any method from the code base used in this experiment. The reason is that appropriate parameter type is not determined in the user queries used for this feature. However, KBCS shows 50%, 33.33%, and 66.67% recall for feature no. 5, 6 and 7 respectively. On the other hand, SBMF shows 100% for features no. 5 and 7, and 33.33% for feature no. 6 as illustrated in Fig. 1. For feature no. 5, two relevant methods are found which names are *transpose* and *rotate* correspondingly. These two methods are feature-wise similar which is detected by SBMF and indexed under common terms (i.e., rotate and transpose). On the other hand, KBCS does not check similarity, and analyzes each method individually during indexing. So, only *rotate* method is retrieved by KBCS. For feature no. 7, SBMF retrieves one more method than KBCS because this method does not contain any term related to *image* but it uses a field of type *Image*. SBMF considers this usage since scaling operation is performed on this field by the method, and adds additional term *Image* against the method.

SBMF, KBCS, and IDCS show equal performance for feature no. 8 (Encoding String to HTML) in terms of recall. However, 50% relevant methods cannot be retrieved because no HTML keyword is found in these method.

Only SBMF is able to retrieve 21 relevant methods whereas other techniques cannot fetch a single method for feature no. 9 (Joining String). Here, SBMF outperforms because it identifies many structurally similar methods which have different names but all these perform string concatenation. Among these, several methods are found which have proper keywords in their body. These keywords are attached to the term list of each similar method by SBMF. As a result, these are indexed under common appropriate terms and all these are retrieved simultaneously. However, other 26 relevant methods cannot be retrieved since no signature matching is found among these.

Number of Retrieved Methods (NRM) and Feature Successfulness Analysis: As NRM is an important measure to perceive recall of a search engine, a comparative result analysis with respect to NRM is performed here. A bar diagram is shown in Fig. 2 depicting feature-wise NRM by SBMF, KBCS, and IDCS. According to the diagram, SBMF retrieves more methods than KBCS and IDCS because of adding common terms to each method.

Although IDCS produces better precision than KBCS and SBMF, it cannot retrieve a single method for some features (such as feature No, 2, 3, 5, 6, 7, 9). The reason is that user queries do not have proper parameter type or return type. This scenario is common when developers have little or no knowledge about the implementation of a feature. KBCS and SBMF mitigate the problem by retrieving more relevant methods adopting free text search. In order to determine whether a feature is successful or not, a metric named feature successfulness is introduced. A feature is said to be successful if at least one relevant method is retrieved that implements the feature. Fig. 3 presents the number of successful features among SBMF, KBCS, and IDCS. According to this figure, SBMF is successful for all the 9 features whereas 7 and 3 successful features are found for KBCS and IDCS respectively. This measure provides a notion that having higher precision is not effective if number of successful feature is low. In addition, improving recall increases the chances of having higher number of successful feature. For this reason, SBMF performs better than KBCS and IDCS.

V. Threats to Validity

In this section, limitations of the experimental study are discussed in terms of internal, external, and construct validity.

a) Internal Validity: In the experiment, there was no control over the skills of the subjects. However, the risks of this threat are reduced by applying repetitive measurement approach because same user created queries for KBCS, IDCS, and SBMF and evaluated the search results.

b) External Validity: The set of features selected may not generalize to the population of software functions. However, these features are among the most common features used for the evaluation in code search. Another possible threat is that projects used in the experiment may not be sufficient enough. However, these projects are statistically representative of open source projects as highlighted in [7].

c) Construct Validity: Existing code clone detection technique can be used to improve recall in code search. However, SBMF differs from code clone detection in several points. SBMF can detect similar methods written in different programming languages and only the execution of method is platform dependent. Another point is that code clone detection technique may provide false positive results to feature-wise clone detection (usually known as Type IV) if values of certain parameters are not defined properly [26]. As a result search engine may retrieve irrelevant methods. However, SBMF checks dynamic behavior through executing method and matches the output for corresponding input to detect feature-wise similar methods. Such mechanism ensures that methods providing the same output, are feature-wise similar and thus no irrelevant method is added to these methods. Besides recall, two other metrics are used in the study to observe the effectiveness of the technique. Although precision is not shown directly in the result analysis due to space limitation, it can be obtained by using data given in TABLE I and Fig. 2.

VI. Conclusion

The recall of a code search engine reduces if similar code fragments are indexed under common proper terms. So, a

technique named SBMF is proposed in this paper which indexes both syntactically and semantically similar methods under common terms. The technique is implemented as a complete software, that constructs index and retrieves relevant methods against the user query.

SBMF first identifies all the methods in the code base by parsing the source code. It converts the methods into reusable methods by resolving data dependency and redefining method signature. Feature-wise similar methods are detected through checking signature and executing methods. Here, methods that produce the same output set for a randomly generated set of input values, are considered as similar methods and these are kept under a cluster. Thus, all the methods are distributed into a set of clusters where each cluster contains feature-wise similar methods and any two clusters differ from one another in implemented feature. All these methods are indexed against the terms that are found in more than half of the methods in the cluster. At last, query expansion is performed to increase the probability of retrieving more methods.

For experimental analysis of the technique, 50 open source projects were selected to build the code base and 9 features were chosen to generate queries. An existing technique named Sourcerer was used to compare the results to SBMF. While analyzing the results it has been seen that SBMF shows 38% improvement in recall than KBCS and 58% than IDCS. It also retrieves relevant methods for all the 9 features, whereas KBCS and IDCS retrieves for 7 and 3 features, respectively. In future, the experiment will be conducted on a large scale dataset to observe the behavior of the technique.

Acknowledgment

This research is supported by ICT Division, Ministry of Posts, Telecommunications and Information Technology, Bangladesh. 56.00.0000.028.33.065.16-747, 14-06-2016.

References

[1] Ruben Prieto-Diaz. Implementing faceted classification for software reuse. *Communications of the ACM*, 34(5):88–97, 1991.

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

[3] Hinrich Schütze. Introduction to information retrieval. In *Proceedings of the international communication of association for computing machinery conference*, 2008.

[4] Randy Smith and Susan Horwitz. Detecting and measuring similarity in code clones. In *Proceedings of the International workshop on Software Clones (IWSC)*, 2009.

[5] Toshihiro Kamiya, Shinji Kusumoto, and Katsuro Inoue. Ccfinder: a multilinguistic token-based code clone detection system for large scale source code. *IEEE Transactions on Software Engineering*, 28(7):654–670, 2002.

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

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

[8] Sushil Bajracharya, Trung Ngo, Erik Linstead, Yimeng Dou, Paul Rigor, Pierre Baldi, and Cristina 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*, pages 681–682. ACM, 2006.

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

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

[11] Suresh Thummalapenta and Tao 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*, pages 204–213. ACM, 2007.

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

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

[14] Otávio Augusto Lazzarini Lemos, Sushil Krishna Bajracharya, and Joel Ossher. 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*, pages 917–918. ACM, 2007.

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

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

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

[18] Wayne C Lim. Effects of reuse on quality, productivity, and economics. *IEEE software*, 11(5):23–30, 1994.

[19] Stefan Haefliger, Georg Von Krogh, and Sebastian Spaeth. Code reuse in open source software. *Management Science*, 54(1):180–193, 2008.

[20] William B Frakes and Brian A Nejmeh. Software reuse through information retrieval. In *ACM SIGIR Forum*, volume 21, pages 30–36. ACM, 1986.

[21] Susan Elliott Sim, Charles LA Clarke, and Richard C Holt. Archetypal source code searches: A survey of software developers and maintainers. In *Program Comprehension, 1998. IWPC'98. Proceedings., 6th International Workshop on*, pages 180–187. IEEE, 1998.

[22] Gordon Fraser and Andrea Arcuri. Sound empirical evidence in software testing. In *Proceedings of the 34th International Conference on Software Engineering*, pages 178–188. IEEE Press, 2012.

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

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

[25] Barbara A Kitchenham, Shari Lawrence Pfleeger, Lesley M Pickard, Peter W Jones, David C. Hoaglin, Khaled El Emam, and Jarrett Rosenberg. Preliminary guidelines for empirical research in software engineering. *IEEE Transactions on software engineering*, 28(8):721–734, 2002.

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