A Search Log Mining based Query Expansion Technique to Improve Effectiveness in Code Search

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 effectiveness of a code search engine is reduced when query terms do not represent the information needs properly or terms are ambiguous. As a result, many irrelevant code snippets and software artifacts are retrieved that hinder the developers reusing existing source code. In this paper, a technique named QExpandator is proposed that improves the effectiveness in code search by expanding query terms with search topic and content specific keywords. It extracts user queries and clicked code fragments from the previous search history and represents each query in a vector document. Jaccard similarity score is calculated for each term in the document vector and a posting list of conceptually similar words is created based on the similarity score. Finally, to expand a user query, top scored terms are retrieved for each query term and appended to the original query. To evaluate the technique, 22 user queries were selected and an existing approach was employed. QExpandator shows 48.6% more effectiveness in terms of precision at 10 (P@10) than the existing one. Moreover, for each query, it increases P@10 from 60.9% to 90% on an average due to using search topic and context specific keywords.**

***Index Terms*—code search, code reuse, term mismatch**

I. Introduction

The effectiveness of a code search engine indicated by precision and recall [1], depends on the accuracy in representing user needs into query terms [2]. Proper transformation of information needs into query terms helps to retrieve more relevant codes. Most of the code search engines provide a single text box to obtain a set of keywords that maps user query [3]. These keywords are matched against the collection index to retrieve relevant code snippets or software artifacts. However, if user provided keywords do not express the desired search topic or context properly, search engines fail to retrieve many relevant codes [4]. Sometimes, many irrelevant code fragments are shown as top ranked search results by search engines due to using irrelevant keywords as query terms. This problem is commonly known as vocabulary mismatch problem in Information Retrieval (IR) [2] and it is also relevant to code search as keywords are used to define user query [5].

In order to increase retrieval effectiveness, vocabulary mismatch problem needs to be solved by expanding query terms. For the expansion, query terms are required to be augmented by new keywords that represent the information needs properly. However, selecting appropriate terms based on the user query is a challenging task. The reason is that user query comprises few keywords and most of these are ambiguous [6]. Another challenge is to match user query with index terms because the indexers and the users do not use the same terms to represent the same code [7]. Even, the probability that two users will use the same query terms to express a particular information needs is 10-15% [7]. Selecting relevant keywords for query expansion requires understanding user intent and searching context. For example, a user provides ”log” as query and expects example codes of logging frameworks that record execution traces of a program. For the same query term ”log”, another user may expect libraries or sample codes that implement different mathematical log functions. Due to this natural language ambiguity, it is also a research challenge to understand the actual intent of a query [8].

Concerned with the increasing demand of reusable software components, researchers have proposed various techniques to improve the effectiveness of code search engines. In order to find example codes about the usage of Application Programming Interface (API), Holmes et al. proposed a technique named Strathcona [9]. It takes keywords from user denoting API name and provides sample code fragments that use the API. However, it cannot retrieve more relevant code fragments if query terms mismatch with API name, because exact matching is performed for code retrieval. Sourcerer [10] provides infrastructure for large code search. It employs traditional IR centric approach to index source codes and query over the index. Although it retrieves software components at different granular level, many irrelevant codes are fetched due to not considering search topic and context. According to the searching behavior, developers like to use keywords as query rather than using concrete object type. CodeGenie employs test case to get semantics from source code and retrieve methods from code base [12]. Although it improves precision in code search, many relevant codes are missed due to using keyword matching without understanding search intent. Lemos et. al. first identified the vocabulary mismatch problem in code search and proposed a thesaurus based query expansion technique to solve the problem [5]. The technique expands a term with its synonyms using WordNet. However, synonyms do not express information needs appropriately when query terms are ambiguous.

In this paper, a technique named QExpandator is proposed to improve the effectiveness in code search. In order to expand query with context and topic specific keywords, the technique adopts user query logs containing the query terms and the code fragments or software artifacts that are clicked. Each user query in the query logs is converted into document vector by the technique. A term-term matrix is constructed where semantically and contextually similar terms are stored in the same row of the matrix. Here, similarity score is calculated by applying Jaccard Similarity formula on term co-occurrences in the same query. At last, for a given user query. top k relevant terms are obtained from the matrix to expand each term. The expanded query is then submitted in the search engine to retrieve relevant code snippets or artifacts.

In order to evaluate the proposed technique, a software tool was developed. Besides, an existing technique named Thesaurus Based Automaic Query Expansion (TBAQE) [5] was also implemented for comparative result analysis. As QExpandator requires query log history, one year search logs were obtained from [13]. 22 real life user queries were selected from Koders (a popular code search engine) and 15 subjects were employed to justify the relevance of the results. Usually, developers use Google to retrieve software artifacts and source code snippets. So, all the queries were executed in Google. To evaluate the effectiveness of QExpandator, a commonly used metric named Precision at 10 (P@10) was employed. While analyzing the results, it is found that TBAQE reduces P@10 by 19.6% due to using synonyms of query terms that do not express the search intent properly. On the other hand, QExpandator increases P@10 from 60.9% to 90% by adding context and topic specific keywords.

II. Related Works

Since the amount of open source codes and software artifacts is increasing rapidly day by day, current development approach suggests reusing existing codes rather than developing the same again. Even, it is said that a significant amount of codes that is written today has already been developed previously. In order to increase reusability, software developers use code search engines to retrieve relevant code fragments and use these as reusable components. Different techniques have been proposed in the literature to improve the effectiveness of these engines. Some significant works are discussed as follows.

A technique named Strathcona assists the developers to understand the usage of an API by retrieving example code snippets [9]. The technique indexes source code fragments based on the API names and takes a set of keywords that represent user query. These keywords are matched with the API names to retrieve relevant example code fragments. However, if query terms mismatch with the API names, the technique cannot retrieve more relevant code fragments because exact matching is performed for code retrieval.

In order to support infrastructure for large scale code search, a technique named Sourcerer is found in the literature [10]. It constructs index of source codes in different granular level such as method, class, package, etc. and search results are retrieved based on the user specified level. Here, traditional IR centric approach is employed to construct index and query over the index but many relevant codes are missed due to not considering search topic and context. As a result, the effectiveness of the technique is reduced.

Lemos et. al. first identified the vocabulary mismatch problem in code search and showed the impact of this in reducing effectiveness of code search engines [5]. A technique was proposed to solve this problem which adopted WordNet for automatic query expansion. The expansion was carried out by adding synonyms of each query term and all these synonyms are joined by boolean OR operation with the corresponding query term. The expanded query is submitted to retrieve relevant code fragments. However, the technique may retrieve many irrelevant codes if query terms are ambiguous. The reason is that synonyms of these terms do not resolve ambiguity by expressing the information needs appropriately.

Reusing existing source codes is a proven approach in the literature for faster and cost effective software development. So, code search engines need to be effective enough to retrieve reusable relevant code fragments that best meet user needs. Various techniques have been found in the literature to improve the effectiveness in code search. However, these techniques suffer from vocabulary mismatch problem that reduces the effectiveness of the code search engines. Although thesaurus based query expansion technique tries to resolve the problem, it does not work well when query terms are ambiguous. So, expanding user query with proper terms to represent information needs appropriately and improve the effectiveness in code search is still an open research issue.

III. Proposed Technique

In order to improve the effectiveness of code search engines, a technique named QExpandator is proposed that expands query terms by mining query logs. The technique constructs a dictionary of conceptually related words by extracting query terms from previous history and ranking based on their co-occurrences. Next, user query is expanded by adding top ranked terms from the dictionary to each query term. The technique comprises three steps which are *Data Preprocessing*, *Term-Term Matrix Construction*, and *Query Reformulation*. Each step is explained in the following subsections.

A. Data Preprocessing

In this step, user query logs are processed to construct a collection of conceptually similar words that acts as data source for query expansion. QExpandator obtains query logs from historical repository and converts each query into document (i.e. collection of terms). Each keyword in the query is checked whether it appears in the retrieved codes that are clicked or downloaded against the query. Keywords that appears at least once are considered relevant, and document is constructed by tokenizing and stemming these keywords.

In Algorithm 1, a complex data type named $QDoc$ is defined which has two fields ($documents$ and $query$) to represent each query obtained from the query log repository. The field $query$ stores query keywords and $documents$ holds source codes that are clicked or downloaded against the corresponding query. The procedure $ConvertQueryToDoc$ takes a list of $QDoc$ and creates a collection of documents from these. A nested *for* loop is employed where the outer *for* loop iterates on $qDocs$ to convert each query into document (Algorithm 1 Line 3-18). For each item in $qDocs$, a list of $String$ ($docKeywords$)

Algorithm 1 Data Processing

```
 1: procedure CONVERTQUERYTODOCUMENT($qDocs$)
 2:     $List < List < String >> docs$
 3:     for each $qDoc \in qDocs$ do
 4:         $List < String > docKeywords$
 5:         for each $doc \in qDoc.documents$ do
 6:             keywords = Process(doc)
 7:             $docKeywords.addAll(keywords)$
 8:         end for
 9:         $List < String > queryTerms$
10:         $queryTerms = Process(qDoc.query)$
11:         $List < String > document$
12:         for each $term \in queryTerms$ do
13:             if $term \in docKeywords$ then
14:                 $document.add(term)$
15:             end if
16:         end for
17:         $docs.add(document)$
18:     end for
19:     return $docs$
20: end procedure
21: procedure PROCESS($List < String > doc$)
22:     $doc = removePunctuation(doc)$
23:     $List < String > processedList, tokens$
24:     $tokens = tokenize(doc)$
25:     for each $token \in tokens$ do
26:         $word = convertToRootForm(token)$
27:         if $word \notin stopWords$ then
28:             $processedList.add(word)$
29:         end if
30:     end for
31:     return $processedList$
32: end procedure
```

Algorithm 2 Term-Term Matrix Construction

```
 1: procedure CONSTRUCTMATRIX($docs$)
 2:     $Map < String, List < String >> matrix$
 3:     $List < String > terms$
 4:     for each $doc \in docs$ do
 5:         for each $term \in doc$ do
 6:             if $term \notin terms$ then
 7:                 $terms.add(term)$
 8:             end if
 9:         end for
10:     end for
11:     for each $t1 \in terms$ do
12:         $List < CandidateTerm > cTerms$
13:         for each $t2 \in terms$ do
14:             if $t1 == t2$ then
15:                 $continue$
16:             end if
17:             $nt1$ = get #Docs Containing Term $t1$
18:             $nt2$ = get #Docs Containing Term $t2$
19:             $nt1Andt2$ = get #Docs Containing $t1$ & $t2$
20:             $nt1Ort2$ = $nt1 + nt2 - nt1Andnt2$
21:             $score = nt1Andt2/nt1Ort2$
22:             $CTerm\ cTerm = new\ CTerm()$
23:             $cTerm.term = t2$
24:             $cTerm.score = score$
25:             $cTerms.add(cTerm)$
26:         end for
27:         $cTerms$ = sort $cTerms$ by score in descending
    order
28:         for each $ct \in cTerms$ do
29:             $matrix[t1].add(ct.term)$
30:         end for
31:     end for
32:     return $matrix$
33: end procedure
```

is defined to store terms from the retrieved codes that are clicked or downloaded. The first inner *for* loop iterates on these codes and invokes another procedure named $Process$ to get terms from each source code. The procedure $Process$ takes source code as input and removes punctuations from the code (Algorithm 1 Line 21-22). After that, it generates tokens and converts each token into term by employing stemming operation. At last, the stop words are removed and a list of all valid terms are returned (Algorithm 1 Line 31). After receiving terms for each $qDocs$, the list $docKeywords$ stores these for pruning irrelevant query terms (Algorithm 1 Line 7). Another list named $queryTerms$ is used to store all the query terms returned by the procedure $Process$ for each query $qDoc.query$. A *for* loop is defined which iterates on $queryTerms$ to find irrelevant terms that are not available in $docKeywords$ (Algorithm 1 Line 12). To generate document for the query $qDoc.query$, all the valid terms are stored in a list named $document$ (Algorithm 1 Line 14). The document is added to the variable $documents$ to create collection of documents which is returned by the procedure $convertToDoc$.

B. Term-Term Matrix Construction

In order to find conceptually similar terms, a Term-Term matrix is constructed from the document collection that is created in the previous step. Each row in the matrix denotes a posting list for a particular term where the list contains all the related terms. Terms that occur in the same query are considered conceptually related to each other. So, these are ranked according to their co-occurrences in the query logs. To expand a particular query term, top ranked terms related to the query term are selected from the list.

The procedure $ConstructMatrix$ in Algorithm 2 takes a list of documents $docs$ obtained from the previous step and constructs collections of conceptually similar terms. A map named $matirx$ is created to store these terms and an empty list $terms$ is declared to save all the terms found in $docs$. A nested *for* loop is defined where the outer loop iterates on every document (doc) in $docs$ and inner loop inserts terms into $terms$ found in the document doc (Algorithm 2 Line 4-10).

Another nested *for* loop is used to calculate similarity score between every pair of terms in $terms$ (Algorithm 2 Line 11-31). For each term $t1$ in $terms$, a list $cTerms$ is declared to contain all the terms conceptually related to $t1$. $CTerm$ is a composite data type which stores term and similarity score by using $term$ and $score$ attributes respectively. In the inner *for* loop, the variables $nt1$ and $nt2$ are initialized with the number of documents that contain $t1$ and the number of documents that contain $t2$ respectively. Another variable $nt1Andt2$ stores the number of documents that contain both $t1$ and $t2$. Next, the number of documents having term $t1$ or $t2$ is calculated and stored in $nt1Ort2$. Later, Jaccard similarity score is calculated by taking ratio between $nt1Andt2$ and $nt1Ort2$ (Algorithm 2 Line 21). A variable $cTerm$ of type $CTerm$ is initialized with term $t2$ and the calculated score. After adding all the terms with respective similarity score, $cTerms$ is sorted by the score in descending order so that higher scored terms can be retrieved easily (Algorithm 2 Line 27). The list is inserted into the map $matrix$ against the term $t1$ (Algorithm 2 Line 28-30). As a result, for a given term, its similar terms can be quickly retrieved from the $matrix$.

C. Query Reformulation

Usually, user query contains few keywords to express user needs. If these keywords are ambiguous, search engines may fail to retrieve many relevant code fragments or fetch irrelevant code snippets. To alleviate this ambiguity and represent information needs more clearly, additional topic specific keywords need to be added to the query. In this step, top scored terms are obtained for each query keyword from the term collection which is constructed in the previous step. Next, user query is reformulated by adding these terms and finally submitted to the search engine as expanded query.

IV. Implementation and Result Analysis

In order to evaluate the proposed technique named QExpandator, a software tool was developed. An existing technique called Thesaurus-based Automatic Query Expansion (TBAQE) [5] was also implemented for comparative analysis. 22 real life user queries were selected from Koders and search logs of one year were collected from [1]. Google is the most widely used search engine to retrieve software components and artifacts. So, all the queries expanded with the techniques were run on Google to compare the results.

A. Environmental Setup

QExpandator was implemented using C# programming language. However, the technique is language and platform independent, and only the fact extraction is language specific. So, it can be implemented in any programming language.

B. Dataset Selection

One year long usage log of a commercial code search engine named Koders was collected from [14]. These dataset was fed into QExpandator to identify topic specific query terms for query expansion. 22 real user queries were obtained from Koders. A summary of these queries are presented in TABLE I where the 2nd column (Query) contains all the original query keywords, and third column (Number of Occurrences) depicts the frequency of the queries within a year.

15 subjects were selected to evaluate the relevance of the search results. 5 of them were senior software engineers and rest 10 were masters students. The reason behind choosing students in this study is that they can play important role in software engineering experiments [15]. All the experimental datasets and source code are available here (http://tinyurl.com/zlfwbjn).

C. Comparative Result Analysis

For comparative result analysis, queries in the experimental dataset were expanded by QExpandator and modified queries were run in Google. Relevance of the retrieved results were judged by the subjects. Same procedure was followed for the existing technique TBQE where queries were expanded using WordNet. To compare the effectiveness between the techniques, a metric named precision at 10 (P@10) was adopted. The reason behind choosing this metric is that the amount of open source projects are increasing rapidly and many relevant code fragments or software artifacts are retrieved by the search engines. Currently, developers are more interested to find the desired results within top 10 retrieved code snippets instead of going through all the fetched results. The metric P@10 is defined as the number of relevant results from the first 10 retrieved code snippets or software artifacts.

A comparative result analysis is depicted in TABLE I where fourth column contains the queries expanded by TBAQE, fifth column contains the queries expanded by QExpandator. For each query, corresponding values of P@10 with respect to orginal query, TBAQE, and QExpandator are shown in fourth, fifth, and sixth columns respectively. While analyzing the results it is seen that 11 queries cannot be expanded by the existing technique TBAQE. The reason is that these queries contain technical terms and names of different frameworks which have no synonyms in the WordNet. On the other hand, QExpandator expands these queries by adding conceptually related terms through analyzing search log history. For example, query#3 contains "awt" which is basically a Graphical User Interface library provided in Java language. So this term is expanded with the most relevant expanded term named "GUI".

Figure 1 depicts the value of P@10 for each query with respect to no expansion (original query), TBAQE, and QExpandator. Here X-axis denotes the query# as shown in TABLE I and Y-axis denotes the value of P@10. According to the figure, TBAQE decreases P@10 for 31.82% of total user queries whereas QExpandator increases P@10 for almost 63.64% user queries and it does not reduce P@10 for the other queries. The reason for such behavior of TBAQE is that it adds synonyms as additional terms to the original query and these terms express different semantic meanings from technical point of view. The expanded query then represents information needs that are different from the original query. For example, in query#19 "sort" keyword is used to get code

TABLE I
COMPARATIVE RESULT ANALYSIS

#	Query	Number of Occurrences	Query Expanded by Theasaurus	Query Expanded by QExpandator (the most relevant term is added)	P@10 for Orginal Query	P@10 for TBAQE	P@10 for Qexpandator
1	apache	411	apache	(apache OR tomcat)	9	9	10
2	audio	360	(audio OR sound)	(audio OR encoder)	4	1	5
3	awt	189	awt	(awt OR GUI)	5	5	10
4	dao	390	dao	(dao OR jdbc)	1	1	10
5	data source	5	(data OR information) AND (source OR seed OR germ OR reference OR beginning OR origin OR root OR rootage OR reservoir OR generator OR author OR informant)	(datasource OR connection)	8	0	8
6	data structure	42	(data OR information) (structure OR construction)	(data structure OR algorithm)	10	9	10
7	date	1840	(date OR appointment OR engagement OR escort OR see)	(date OR format)	6	1	6
8	files	2164	files	(files OR class)	2	2	10
9	ftp	1865	ftp	(ftp OR server)	10	10	10
10	hibernate	640	hibernate	(hibernate OR jpa)	10	10	10
11	huffman	967	huffman	(huffman OR coding)	7	7	10
12	image	1693	(image OR effigy OR simulacrum OR picture OR icon OR ikon OR persona OR prototype OR paradigm OR epitome OR trope OR figure OR double OR visualize OR visualiseOR envision OR project OR fancy OR see)	(image OR attribute)	0	0	5
13	jsp	423	jsp	(jsp or java)	8	8	10
14	list	1241	(list OR tilt OR inclination OR lean OR leaning OR listing OR name OR number OR heel)	(list or util)	8	0	10
15	listener	208	(listener OR hearer OR auditor OR attender)	(listener or event)	0	0	10
16	log	879	(log OR logarithm OR backlog OR lumber)	(log OR log4j)	3	0	10
17	lucene	383	lucene	(lucene OR solr)	10	10	10
18	parser	1449	parser	(parser OR dom)	7	7	10
19	sort	2402	(sort OR kind OR form OR variety OR sorting OR classify OR class OR assort OR separate OR screen OR sieve)	(sort OR bubble)	10	0	10
20	spring	381	(spring OR leap OR leaping OR saltation OR bound OR bounce OR give OR springiness OR fountain OR outflow OR outpouring OR springtime OR resile OR rebound OR recoil OR reverberate OR ricochet OR jump OR form)	(spring OR framework)	5	0	7
21	test	1537	(test OR trial OR run OR tryout OR examination OR exam OR quiz OR prove OR try OR examine OR essay OR screen)	(test OR Junit)	1	1	7
22	webservice	297	webservice	(webservice OR SOAP)	10	10	10

snippets that order items in a list in specific order. One of the synonyms of sort is "sieve" which is used in the expanded query by TBAQE. "sieve" represents a prime number generator algorithm in programming context and it has no similarity with the keyword "sort". As a result, the expanded query suffers from reduced P@10. Conversely, "bubble" keyword is appended to the query sort by QExpandator and it is consistent with the search topic because bubble sort is a special type of sorting algorithm. Due to adding context and topic specific terms, QExpandator increases the value of P@10.

Ambiguity in query terms are found in several queries such as query#2, query#14, query#15, query#16, and query#21 as shown in TABLE I. Usually ambiguous terms express different semantic meanings in different contexts. For example, in query#16 "log" keyword has three semantic meanings which are logarithmic function, an element of a tree, and observing execution traces of a program. TBAQE uses "logarithm" as synonym of "log" keyword and forces the search engine to retrieve logarithmic functions which reduces P@10. This is because the search intent is to find example code of tracking execution behavior. However, QExpandator adds "log4j" as expanded term which is relevant to search topic. The technique handles ambiguous terms by adding search intent specific keywords and increases the effectiveness of search engine.

Although QExpandator improves P@10 for 63.64% of total user queries by adding search topic specific keywords, and for the rest 36.36% queries, it cannot increase the value of P@10. The reason is that most of these queries contain keywords that represent the information needs properly. Conversely, TBAQE reduces P@10 for 31.82% user queries due to adding only synonyms that do not consider search context properly.

In essence, QExpandator improves P@10 by 29.1% on an average that leads to the ultimate P@10 value 90% for each user query. Moreover, the technique does not hurt or reduce the value of P@10 for any query. Conversely, TBAQE reduces the P@10 for 31.82% user queries and no improvement is found in any user query for this technique. QExpandator performs better than TBAQE because it adopts search topic and context specific keywords for query expansion instead of synonyms that have different meanings in various contexts.

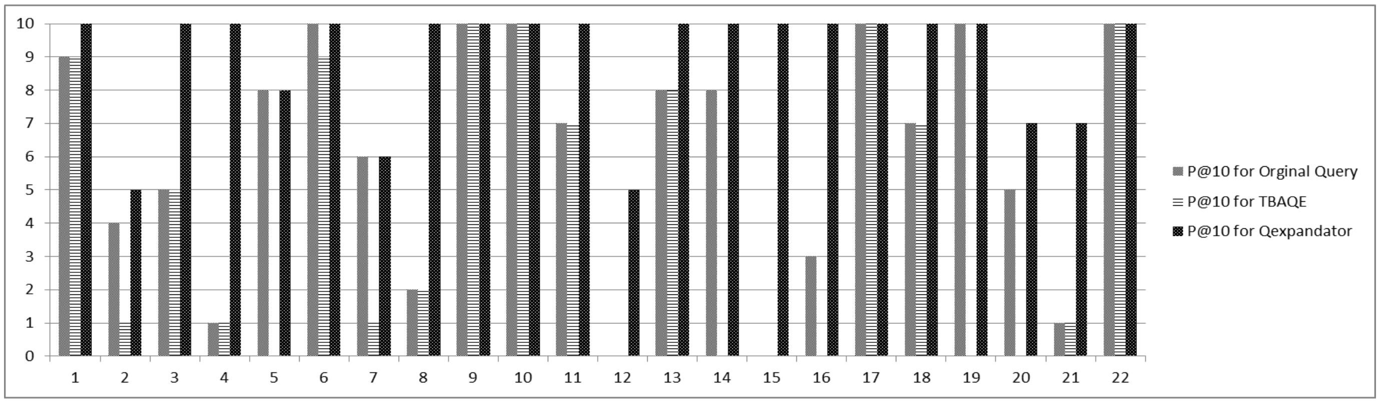

Fig. 1. Precision at 10 Analysis

V. Conclusion

The effectiveness in code search is reduced when query terms do not express the desired information needs properly. As a result, many relevant code fragments are missed and irrelevant code snippets are retrieved against user query. In this paper, a technique named QExpandator is proposed which increases the effectiveness in code search by expanding user query with search topic and context specific keywords.

QExpandator adopts search log history to identify conceptually similar words. It converts each query into document vector where each term in the vector corresponds to the query term. Conceptually similar terms are identified by employing Jaccard similarity on the co-occurrence of document terms. A term-term matrix is constructed to store all the similar terms in the same row in descending order of the similarity score. Finally, for a given user query, each query term is expanded by adding top scored terms from the matrix.

In order to evaluate the proposed technique, a software tool was developed and 22 user queries were used for comparative analysis with the existing technique TBAQE. Precision at 10 was used as evaluation metric for its popularity in effectiveness measurement. The result analysis demonstrates that QExpandator produces 48.6% more improvement in P@10 than TBAQE. Moreover, on an average, it increases the value of P@10 from 60.9% to 90% for each query. The reason is that instead of using synonyms, it adds context specific keywords as expanded terms. In future, more real life user queries will be used 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] 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 *15th International Working Conference on Source Code Analysis and Manipulation (SCAM)*, pages 41–50. IEEE, 2015.

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

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

[4] Claudio Carpineto and Giovanni Romano. A survey of automatic query expansion in information retrieval. *ACM Computing Surveys (CSUR)*, 44(1):1–56, 2012.

[5] 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.

[6] Tessa Lau and Eric Horvitz. Patterns of search: analyzing and modeling web query refinement. In *Proceedings of the 7th International Conference on User Modeling*, pages 119–128. Springer-Verlag, 1999.

[7] George W. Furnas, Thomas K. Landauer, Louis M. Gomez, and Susan T. Dumais. The vocabulary problem in human-system communication. *Communications of the ACM*, 30(11):964–971, 1987.

[8] George A Miller. Wordnet: a lexical database for english. *Communications of the ACM*, 38(11):39–41, 1995.

[9] 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.

[10] 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.

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

[12] 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.

[13] Sushil Krishna Bajracharya and Cristina Videira Lopes. Mining search topics from a code search engine usage log. In *Proceedings of the 6th IEEE International Working Conference on Mining Software Repositories*, pages 111–120. Citeseer, 2009.

[14] 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.

[15] 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.