Query Expansion via Wordnet for Effective Code Search

Meili Lu*, Xiaobing Sun*†, Shaowei Wang‡, David Lo‡ Yucong Duan§
*School of Information Engineering, Yangzhou University, Yangzhou, China
‡School of Information Systems, Singapore Management University, Singapore
§School of Information Science and Technology, Hainan University, Haikou, China
†State Key Laboratory for Novel Software Technology, Nanjing University, Nanjing, China

Abstract*—Source code search plays an important role in software maintenance. The effectiveness of source code search not only relies on the search technique, but also on the quality of the query. In practice, software systems are large, thus it is difficult for a developer to format an accurate query to express what really in her/his mind, especially when the maintainer and the original developer are not the same person. When a query performs poorly, it has to be reformulated. But the words used in a query may be different from those that have similar semantics in the source code, i.e., the synonyms, which will affect the accuracy of code search results. To address this issue, we propose an approach that extends a query with synonyms generated from WordNet. Our approach extracts natural language phrases from source code identifiers, matches expanded queries with these phrases, and sorts the search results. It allows developers to explore word usage in a piece of software, helps them quickly identify relevant program elements for investigation or quickly recognize alternative words for query reformulation. Our initial empirical study on search tasks performed on the JavaScript/ECMAScript interpreter and compiler, Rhino, shows that the synonyms used to expand the queries help recommend good alternative queries. Our approach also improves the precision and recall of *Conquer*, a state-of-the-art query expansion/reformulation technique, by 5% and 8% respectively.

I. Introduction

Software change is a fundamental ingredient of software maintenance [1], [2]. These changes occur due to the continuous change requests proposed by end users. Given a change request, developers or maintainers need to analyze the change request, and locate the parts in a software system which are related to the change request. To help developers perform this task, a number of text retrieval (*TR*)-based code search approaches have been proposed [3]–[8]. These approaches take as input the change request as a query, and return a ranked list of program elements (e.g., methods, files, etc.) that match the query.

A common issue with all *TR*-based code search approaches is that the results of the retrieval depend greatly on the quality of the change query. However, as software systems keep evolving and growing, locating code for software maintenance becomes increasingly difficult. The words used in a query written by a maintainer can be different from the lexicon used by the developers. Thus, the maintainer will spend much time and energy to rewrite a query many times, which affects the effectiveness of a code search tool to aid software maintenance tasks.

When a maintainer gives a poor query, the query needs to be reformulated. Many existing code search techniques provide little support to help developers rewrite a query [3]–[6]. Most of the code search techniques simply list search results as a list of files, method signatures, or lines of code where the query matched (e.g., *Eclipse* file search or grep), and display the results by decreasing relevance. Since a good query is very important to improve search results, and a poor query can waste developers' lots of time, there is a need for a code search tool that can help users reformulate queries. This need has motivated researchers to develop a number of query reformulation approaches, e.g., [7], [9], [10]. These approaches are not perfect yet and additional improvements are needed.

In this paper, to improve existing query reformulation techniques, we introduce a novel approach to reformulate a query. We implement an approach that can expand a query by using synonyms that are obtained from WordNet [11]. Our approach first identifies the part-of-speech (*POS*) of each word that makes up the query and finds the synonyms of each word with the same *POS* using WordNet. Then, it expands the original query by replacing the words of the query with their corresponding synonyms in WordNet [11]. Next, it extracts natural language phrases from source code identifiers of methods in a code base, and matches these phrases against the expanded queries to get the search results (i.e., relevant methods) that are sorted based on their similarities with the expanded queries.

We performed a preliminary evaluation using 19 search tasks that were used by Hill et al. to evaluate an existing query reformulation tool [9]. The results show that the synonyms used to expand the query help recommend good alternative queries. Our approach also outperforms *Conquer*, the latest query reformulation technique, by Hill et al. [9].

The rest of our paper is organized as follows. Section 2 introduces the background of Wordnet. We describe the details of our proposed approach in Section 3. A preliminary evaluation of our approach is presented in Section 4. In Section 5, some related work is discussed. Finally, we conclude our paper and mention future work in Section 6.

II. Background

Our approach implements query expansion with synonyms with the help of Wordnet [11]. Wordnet is a useful tool for

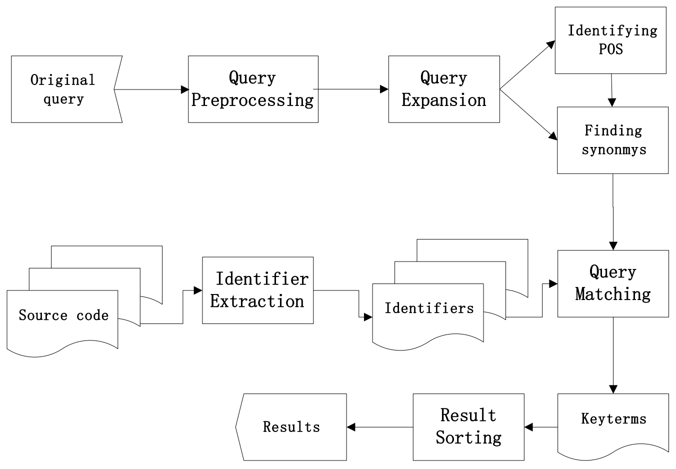

Fig. 1. Process of our approach

computational linguistics and natural language processing. It resembles a thesaurus, in which it groups words together based on their meanings. Nouns, verbs, adjectives and adverbs are grouped into sets of cognitive synonyms (i.e.,synsets), each expressing a distinct concept. Synsets are interlinked by means of conceptual-semantic and lexical relations. However, there are some important distinctions. First, Wordnet interlinks not just word forms—strings of letters—but specific senses of words. As a result, words that are found in close proximity to one another in the network are semantically disambiguated. Second, Wordnet labels the semantic relations among words, whereas the groupings of words in a thesaurus does not follow any explicit pattern other than meaning similarity. When we pass the two arguments (the word and its POS) to Wordnet, we can get a set of its synonyms. More detailed information about Wordnet is available online[1].

III. Approach

The process that our approach follows is shown in Figure 1. It has five steps, query preprocessing, query expansion, identifier extraction, query matching, and result sorting. At the end of the process, it produces a list of results made up of signatures of methods related to the query and alternative phrases extracted from source code identifiers of the methods. The following subsections describe the details of each step of our approach.

1) Query Preprocessing: We perform standard text preprocessing steps on the query. We first remove stop words, i.e., words that appear very often and does not have definite meaning, for example, prepositions and auxiliary words such as *the*, *it*, *in*, etc. Then, we perform stemming, camel case and underscores splitting to get a set of words [12]. For example, consider an original query "*Converts decimal to hexadecimal*". After preprocessing, we get the set of words {*convert, decimal, hexadecimal*}.

[1]http://wordnet.princeton.edu/wordnet/

2) Query Expansion: After the query has been preprocessed, we expand it in the following way. First, we identify the parts-of-speech (*POS*) of each word in the preprocessed word set. The *POS* of each word can be easily identified using a *POS* Tagger [13]. A *POS* Tagger is a tool that reads text and assigns *POS* to each word.

Second, we identify the synonyms of each word to expand the original query using *WordNet*, which is a useful tool for computational linguistics and natural language processing. If we use all of the synonyms of a word, some inappropriate synonyms are returned by Wordnet [14]. We deal with this problem in this way. We only return synonyms of a word in WordNet that have the same *POS*. Thus, the possibility that each of these synonyms to be inappropriate will be decreased. We replace words in the original query with their synonyms to create expanded queries. For example, supposing that a query is "*display lyrics*", the synonyms for "display" (with the same which *POS*) are {*expose, exhibit, reveal, show*}, and the synonyms for "lyrics" (with the same which *POS*) are {*words, language, poem*}, after the query expansion step we get a set of 20 phrases (or expanded queries), which is {*display lyrics, expose lyrics, exhibit lyrics, reveal lyrics, reveal lyrics, display words, display language, display lyric poem, expose words, expose language, expose lyric poem, exhibit words, exhibit language, exhibit lyric poem, reveal words, reveal language, reveal lyric poem, show words, show language, show lyric poem*}.

3) Identifier Extraction: The above two subsections focus on the query. In this step, we focus on the methods in the source code. For each method, we create its representative set of words by extracting identifiers from methods. Identifiers are an important source of domain information and can often serve as a starting point in many program comprehension tasks [15].

To extract the identifiers, we first create a set of Java reserved keywords and remove these keywords from the source code. We then use space, numbers, and other delimiters, e.g.,".", ";", to separate one identifier from the others. For example, we can extract identifiers {*eb* , *cstop* , *swing*} from the following piece of code: "*package eb.cstop.swing*;". In addition, we also perform camel case and underscores splitting. For instance, "*CrystalPlayer*" is split into "*crystal player*" and "*Decimal2Hex*" is split into "*decimal hex*".

4) Query Matching: In this step, we match the expanded queries with the identifiers. For each expanded query $eQuery$ and each identifier id we compute the following similarity score:

$$Sim(eQuery, id) = \frac{|eQuery \bigcap id|}{max(|eQuery|, |id|)}$$

In the above equation, *max(*$|eQuery|$, $|id|$ represents the largest number of words in $eQuery$ and id, while $|eQuery \bigcap id|$ is the number of common words shared between the two sets. For instance, given an expanded query "*display lyrics*", its similarity with the identifier "*get lyrics list*" is 1/3. If the similarity score is larger than the threshold set by developers, we regard the identifier as a *keyterm* and forward it to the next step.

5) Result Sorting: Once the *keyterms* have been identified, we want to identify methods that are likely to be related to the original query. We do this by checking for the appearance of each *keyterm* in a method.

Let us denote the total number of all *keyterms* as *sum*, and the number of *keyterms* in each method in a code base as n_1, n_2, $\cdots$, n_k, where k is the total number of methods in the code base. Let us also use p_1, p_2, $\cdots$, p_k to represent the percentage of *keyterms* that appear in each method. That is, p_i=n_i/sum.

We sort the methods in the code base based on their corresponding percentages in a descending order. The higher the percentage, our approach deems the method to be more relevant. As a final result, we present each of the methods in the following format:

$$method\ signature = \{identifiers\ in\ the\ method\}$$

For the query "*display lyrics*", one of the results is: "*public void lyricsDowned (LyricsContents contents)* = { *get lyrics list, lyrics writer, is show lyrics, crystal lyrics list, set lyrics, lyrics content, get lyrics, lyrics list, show select list, lyrics selecter, lyrics downed, add lyrics, lyrics item, lyrics contents*}", where the phrases in braces, i.e., {}, separated by commas are the natural language identifiers extracted from the source code of the method. The phrases extracted from method identifiers can help developers to quickly determine whether the method is relevant or not and reformulate a better query.

IV. Preliminary Evaluation

We conducted a preliminary empirical study to investigate the performance of our approach. In this study, we asked 20 volunteer Java developers with varying levels of programming skills to perform 19 search tasks. These tasks are performed on a 45 KLOC JavaScript/ECMAScript interpreter and compiler, Rhino, following the details provided by Hill et al. to evaluate their query reformulation approach [9]. Each search task maps to a feature described by a subsection of Rhino's documentation. Each search task corresponds to a screen shot of the feature being executed. This screen shot is given to the participants to formulate queries to search for code that implements the feature.

There are several aspects of our approach that we want to evaluate in our preliminary study. First, we want to investigate whether our query expansion helps improve the results. Second, we want to assess whether our approach performs better than the state-of-the-art query expansion (aka. query reformulation) technique *Conquer* [9]. *Conquer* is an approach that automatically extracts natural language phrases from source code identifiers and categorizes the phrases and search results in a hierarchy [9]. It requires a user to enter a query as input and the results are composed of four components: (1) prevalence of query words, (2) suggested alternative query words, (3) categorization by action or theme, and (4) phrase list. Prevalence of query words is the frequency of each query word in the relevant results. Suggested alternative query words suggest alternative query words that appear in the source code. The results of *Conquer* will be divided into two hierarchies organized by action and theme. *Conquer* also returns a phrase list composed of phrases, methods and files. For our approach, we set the similarity score threshold as 3/7 based on our empirical investigation in our studies.

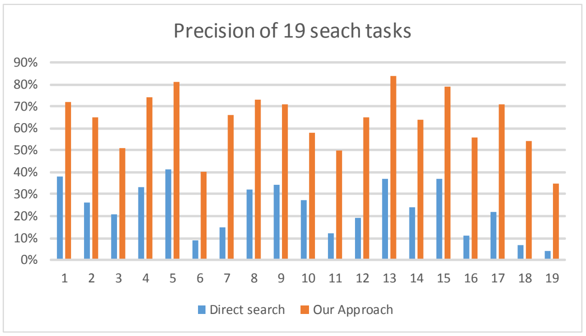

Fig. 2. Precision of direct search (i.e., no query expansion) and our approach on 19 search tasks

As discussed above, we evaluated our approach from two aspects. First, we compared the performance of the expanded queries created by our approach with that of the original queries. Second, we compared the performance of our approach with *Conquer* to see whether our approach outperforms the state-of-the-art. We measure performance in terms of search effectiveness. To evaluate search effectiveness, the participants in our user study answer the following two survey questions (*SQs*) about each search task that they have completed.

SQ1: How many results returned by a query expansion technique are relevant?

SQ2: How many relevant results are not returned by a query expansion technique?

Search effectiveness is then measured based on the answers to these two survey questions by calculating the precision and recall of each search result [16], [17]. Precision is the percentage of search results that are relevant, and captures how many irrelevant results were present with the relevant results. Recall is the percentage of all relevant results that were correctly returned as search results, and captures how many of the actually relevant results were predicted as relevant. We use the following two formulas to calculate the precision and recall of the search tasks.

$$Precision = \frac{number\quad of\quad relevant\quad results\quad retrieved}{number\quad of\quad results\quad retrieved}$$
$$Recall = \frac{number\quad of\quad relevant\quad results\quad retrieved}{number\quad of\quad relevant\quad docs}$$

A. Results

No query expansion vs. Query expansion by our approach. For the 19 search tasks, all the participants reflected that the synonyms used to expand the query help recommend good alternative queries. Since the participants were not familiar

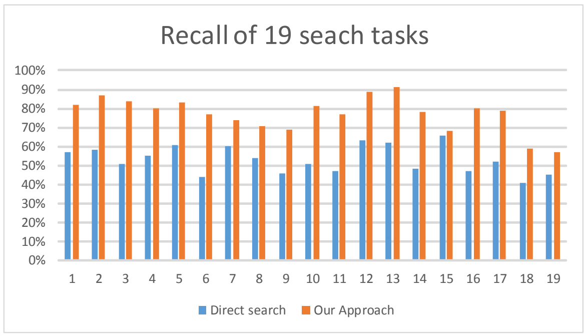

Fig. 3. Recall of direct search (i.e., no query expansion) and our approach on 19 search tasks

TABLE I
PRECISION OF OUR APPROACH AND *Conquer*

Task	**Techniques**					
	Our approach			**Conquer**		
	min	*max*	*mean*	*min*	*max*	*mean*
1	**64%**	**87%**	72%	62%	84%	72%
2	**56%**	76%	65%	55%	77%	67%
3	**45%**	**67%**	**51%**	43%	58%	49%
4	**66%**	**83%**	**74%**	56%	76%	67%
5	**76%**	**88%**	81%	75%	87%	81%
6	34%	47%	40%	43%	54%	47%
7	**54%**	**78%**	**66%**	45%	67%	56%
8	**67%**	**79%**	**73%**	37%	48%	45%
9	61%	83%	71%	77%	83%	80%
10	54%	62%	58%	63%	69%	65%
11	47%	55%	50%	74%	78%	76%
12	56%	76%	65%	71%	83%	76%
13	**77%**	**89%**	**84%**	56%	71%	64%
14	**59%**	**75%**	**64%**	54%	68%	59%
15	**76%**	**83%**	**79%**	68%	79%	74%
16	**44%**	**67%**	**56%**	33%	45%	37%
17	**66%**	**79%**	**71%**	35%	47%	43%
18	**51%**	**60%**	**54%**	26%	37%	29%
19	23%	**45%**	**35%**	31%	37%	33%

with *Rhino*, it was very difficult for them to write a good query at the beginning. Most of the queries they wrote at the beginning returned no results or the results did not meet their original intention. What's worse, without query expansion by our approach, the results returned could not give any hints for participants to rewrite a query.

Our approach solved the above-mentioned problem. The precision and recall results of direct search (i.e., no query expansion) and our approach are shown in Figure 2 and 3. From the figures, we can note that the precision and recall results of our approach are better than those of direct search. On average, our approach improves the precision and recall of direct search by 40% and 24%, respectively. This indicates that the alternative queries recommended by our approach could reflect the true intention of the developers. Although there were some results that were not relevant, overall our approach greatly helped the participants rewrite a good query close to the ideal query. Hence, instead of disturbing developers, the expanded query recommended by our approach help developers formulate better queries.

***Conquer* vs. Our approach.** In addition, we also conducted

TABLE II
RECALL OF OUR APPROACH AND *Conquer*

Task	**Techniques**					
	Our approach			**Conquer**		
	min	*max*	*mean*	*min*	*max*	*mean*
1	**77%**	**86%**	**82%**	67%	79%	72%
2	**81%**	**93%**	**87%**	76%	82%	79%
3	**82%**	**89%**	**84%**	65%	77%	72%
4	**78%**	**83%**	**80%**	56%	71%	64%
5	**81%**	84%	**83%**	75%	87%	81%
6	76%	79%	77%	76%	88%	81%
7	**71%**	**79%**	**74%**	34%	45%	41%
8	**69%**	**74%**	**71%**	56%	67%	61%
9	**65%**	78%	**69%**	54%	78%	64%
10	67%	86%	81%	81%	89%	85%
11	70%	85%	77%	73%	91%	85%
12	**87%**	**93%**	**89%**	72%	78%	75%
13	**86%**	**96%**	**91%**	71%	84%	79%
14	**76%**	**79%**	**78%**	67%	74%	68%
15	**61%**	**74%**	**68%**	45%	56%	49%
16	**76%**	**85%**	**80%**	42%	56%	49%
17	**77%**	**83%**	**79%**	67%	76%	71%
18	56%	64%	59%	74%	84%	78%
19	45%	65%	57%	76%	79%	77%

a comparison between our approach and the state of the art technique, *Conquer* [9]. Tables I and II report the precision and recall scores of our approach and *Conquer* on the 19 search tasks, respectively. In the tables, *min*, *max*, and *mean* denote the *minimum*, *maximum* and *mean* values of the respective score. The results show that the precision and recall of our approach are higher than *Conquer* for many search tasks in spite of a few of exceptions. For example, for the fourth search task in Table I, the minimum, maximum and mean precision scores of our approach are 66%, 83%, 74%, respectively; On the other hand, the scores for *Conquer* are 56%, 76%, 67%, respectively. For the fourth search task in Table II, the minimum, maximum and mean recall scores of our approach are 78%, 83%, 80%, and the scores are 56%, 71%, 64% for *Conquer*. On average, across all the tasks, our approach outperforms *Conquer*'s precision and recall by 5% and 8%, respectively. Hence, we can conclude that our proposed approach performs better than *Conquer* search technique.

V. RELATED WORK

Query expansion (or reformulation) has long been established as a way to improve the results returned by a *TR* engine [18]. Some existing software engineering work have been focusing on this task to help code search. We highlight some of them below.

Shepherd et al. developed a tool, *V-DO*, which automatically extracts *V-DO* (verb - direct object) pairs from source code comments and identifiers for query recommendations [19]. *V-DO* automatically suggests close matches for misspelled query terms [19]. Haiduc et al. proposed an approach that can recommend a good query reformulation strategy by performing machine learning on a set of historical queries and relevant results [10]. It suggests four strategies to formulate a query, including three query expansion techniques and one reduction techniques. Recently, Hill et al. proposed a query expansion tool named *Conquer*, which automatically extracts natural language phrases from source code identifiers and categorizes the phrases and search results in a hierarchy [9]. *Conquer* combines *V-DO* [19] and contextual search technique [20]. It introduces a novel natural language based approach to organize and present search results and suggest alternative query words. In our approach, we expand the original query with synonyms from WordNet and extract natural language phrases from source code identifiers. The results of our approach are composed of method signatures corresponding to the expansion of the original query and alternative phrases extracted from source code identifiers. Our preliminary empirical results show that our approach improves *Conquer* in terms of precision and recall.

VI. Conclusion and Future Work

Source code retrieval plays an important role in many software engineering tasks, especially in software maintenance tasks. However, designing a query that can accurately retrieve the relevant software artifacts is challenging for developers as it requires a certain level of knowledge and experience regarding the code base. In this paper, we propose a new approach which implements query expansion using synonyms, extracts natural phrases from source code identifiers, matches the expanded queries with identifiers, and sorts methods in a code base based on the matched identifiers. The results returned by our approach are composed of method signatures and the alternative phrases extracted from method identifiers, which can help developers to quickly determine whether the results are relevant or not and reformulate a better query. Our preliminary evaluation results show that our approach helps recommend better reformulations. Moreover, our approach outperforms *Conquer*, a state-of-the-art query expansion technique, in terms of precision and recall.

In the future, we want to conduct more experiments using additional search tasks performed on various software projects written in various programming languages to evaluate the generality and effectiveness of our approach.

Acknowledgment

The authors would like to thank the 20 participants who spend time and effort completing our evaluation. This work is supported partially by Natural Science Foundation of China under Grant No. 61402396 and No. 61472344, partially by the Natural Science Foundation of the Jiangsu Higher Education Institutions of China under Grant No. 13KJB520027, partially by the Open Funds of State Key Laboratory for Novel Software Technology of Nanjing University under Grant No. KFKT2014B13, partially by the Jiangsu Training Programs of Innovation and Entrepreneurship for Undergraduates under Grant No. 201411117066X, and partially by the Cultivating Fund for Science and Technology Innovation of Yangzhou University under Grant No. 2013CXJ025.

References

[1] V. Rajlich, "Software evolution and maintenance," in *Proceedings of the on Future of Software Engineering*, ser. FOSE 2014, 2014, pp. 133–144.

[2] G. Canfora, L. Cerulo, M. Cimitile, and M. Di Penta, "How changes affect software entropy: An empirical study," *Empirical Softw. Engg.*, vol. 19, no. 1, pp. 1–38, Feb. 2014.

[3] D. Poshyvanyk, M. Petrenko, A. Marcus, X. Xie, and D. Liu, "Source code exploration with google," in *22nd IEEE International Conference on Software Maintenance (ICSM 2006), 24-27 September 2006, Philadelphia, Pennsylvania, USA*, 2006, pp. 334–338.

[4] T. B. Le, S. Wang, and D. Lo, "Multi-abstraction concern localization," in *2013 IEEE International Conference on Software Maintenance, Eindhoven, The Netherlands, September 22-28, 2013*, 2013, pp. 364–367.

[5] W. Chan, H. Cheng, and D. Lo, "Searching connected API subgraph via text phrases," in *20th ACM SIGSOFT Symposium on the Foundations of Software Engineering (FSE-20), SIGSOFT/FSE'12, Cary, NC, USA - November 11 - 16, 2012*, 2012, p. 10.

[6] S. Wang, D. Lo, Z. Xing, and L. Jiang, "Concern localization using information retrieval: An empirical study on linux kernel," in *18th Working Conference on Reverse Engineering, WCRE 2011, Limerick, Ireland, October 17-20, 2011*, 2011, pp. 92–96.

[7] L. L. Pollock, K. Vijay-Shanker, E. Hill, G. Sridhara, and D. Shepherd, "Natural language-based software analyses and tools for software maintenance," in *Software Engineering - International Summer Schools, ISSSE 2009-2011, Salerno, Italy. Revised Tutorial Lectures*, 2011, pp. 94–125.

[8] B. Dit, M. Revelle, M. Gethers, and D. Poshyvanyk, "Feature location in source code: a taxonomy and survey," *Journal of Software: Evolution and Process*, vol. 25, no. 1, pp. 53–95, 2013.

[9] E. Hill, M. Roldan-Vega, J. A. Fails, and G. Mallet, "Nl-based query refinement and contextualized code search results: A user study," in *2014 Software Evolution Week - IEEE Conference on Software Maintenance, Reengineering, and Reverse Engineering, CSMR-WCRE 2014, Antwerp, Belgium, February 3-6, 2014*, 2014, pp. 34–43.

[10] S. Haiduc, G. Bavota, A. Marcus, R. Oliveto, A. D. Lucia, and T. Menzies, "Automatic query reformulations for text retrieval in software engineering," in *35th International Conference on Software Engineering, ICSE '13, San Francisco, CA, USA, May 18-26, 2013*, 2013, pp. 842–851.

[11] C. Leacock and M. Chodorow, "Wordnet: An electronic lexical database - combining local context and wordnet similarity for word sense identification, in wordnet: An electronic lexical database," 1998, pp. 265–283.

[12] X. Sun, X. Liu, J. Hu, and J. Zhu, "Empirical studies on the nlp techniques for source code data preprocessing," in *Proceedings of the 2014 3rd International Workshop on Evidential Assessment of Software Technologies*, ser. EAST 2014, 2014, pp. 32–39.

[13] K. Toutanova and C. D. Manning, "Enriching the knowledge sources used in a maximum entropy part-of-speech tagger," in *Proceedings of the Joint SIGDAT Conference on Empirical Methods in Natural Language Processing and Very Large Corpora (EMNLP/VLC-2000)*, 2000, pp. 63–70.

[14] G. Sridhara, E. Hill, L. L. Pollock, and K. Vijay-Shanker, "Identifying word relations in software: A comparative study of semantic similarity tools," in *The 16th IEEE International Conference on Program Comprehension, ICPC 2008, Amsterdam, The Netherlands, June 10-13, 2008*, 2008, pp. 123–132.

[15] T. Fritz, G. C. Murphy, E. R. Murphy-Hill, J. Ou, and E. Hill, "Degree-of-knowledge: Modeling a developer's knowledge of code," *ACM Trans. Softw. Eng. Methodol.*, vol. 23, no. 2, p. 14, 2014.

[16] C. D. Manning, P. Raghavan, and H. Schütze, *Introduction to Information Retrieval*. New York, NY, USA: Cambridge University Press, 2008.

[17] C. D. Manning, P. Raghavan, and H. Schütze, *Introduction to information retrieval*. Cambridge University Press, 2008.

[18] J. Rocchio, "Relevance feedback in information retrieval," 1971.

[19] D. Shepherd, Z. P. Fry, E. Hill, L. L. Pollock, and K. Vijay-Shanker, "Using natural language program analysis to locate and understand action-oriented concerns," in *Proceedings of the 6th International Conference on Aspect-Oriented Software Development, AOSD 2007, Vancouver, British Columbia, Canada, March 12-16, 2007*, 2007, pp. 212–224.

[20] E. Hill, L. L. Pollock, and K. Vijay-Shanker, "Automatically capturing source code context of nl-queries for software maintenance and reuse," in *31st International Conference on Software Engineering, ICSE 2009, May 16-24, 2009, Vancouver, Canada, Proceedings*, 2009, pp. 232–242.