Automated Construction of a Software-Specific Word Similarity Database

Yuan Tian[1], David Lo[1], and Julia Lawall[2]
[1]Singapore Management University, Singapore
[2]Inria/LIP6, Regal
{yuan.tian.2012,davidlo}@smu.edu.sg, julia.lawall@lip6.fr

***Abstract*—Many automated software engineering approaches, including code search, bug report categorization, and duplicate bug report detection, measure similarities between two documents by analyzing natural language contents. Often different words are used to express the same meaning and thus measuring similarities using exact matching of words is insufficient. To solve this problem, past studies have shown the need to measure the similarities between pairs of words. To meet this need, the natural language processing community has built WordNet which is a manually constructed lexical database that records semantic relations among words and can be used to measure how similar two words are. However, WordNet is a general purpose resource, and often does not contain software-specific words. Also, the meanings of words in WordNet are often different than when they are used in software engineering context. Thus, there is a need for a software-specific WordNet-like resource that can measure similarities of words.**

In this work, we propose an automated approach that builds a software-specific WordNet like resource, named $WordSim^{SE}_{DB}$, by leveraging the textual contents of posts in StackOverflow. Our approach measures the similarity of words by computing the similarities of the weighted co-occurrences of these words with three types of words in the textual corpus. We have evaluated our approach on a set of software-specific words and compared our approach with an existing WordNet-based technique ($WordNet^{res}$) to return top-k most similar words. Human judges are used to evaluate the effectiveness of the two techniques. We find that $WordNet^{res}$ returns no result for 55% of the queries. For the remaining queries, $WordNet^{res}$ returns significantly poorer results.

I. Introduction

During the development of software systems, developers and other stakeholders often create natural language artifacts to communicate with one another. Subsequently, developers often need to analyze these natural language artifacts to perform various software engineering tasks. Several studies have proposed automated means to improve these tasks. For example, code search takes in a natural language query and returns code fragments that correspond to the query [16], duplicate bug report detection identifies reports that describe the same issue but are written in different ways by different reporters [30], etc. The basic task involved in these techniques is measuring the similarity among two natural language artifacts (documents).

In natural language documents, different words can be used to express the same meaning. Thus, to measure the similarity of two documents, exact matching of words is insufficient. There is a need to capture the semantic distance between two words. For example, words such as paper, pen, and pencil are more semantically similar to one another than computer, mountain, and Jupiter. Measuring semantic distance of words is natural for humans, but it is much harder for machines. The natural language processing community has long investigated the problem of measuring the similarity of two words [14], [15]. Understanding the similarity of words makes it possible to better measure the similarity of documents. Measuring word similarity has been shown to improve machine learning tasks, e.g., information retrieval [5], text categorization [12], etc. Many automated software engineering problems can be mapped to these machine learning tasks, e.g., [19], [30]. These problems can be improved by levering word similarity information. Indeed, several research works have shown that similar words can be used to expand or refine search queries when applying code search [18], [29]. Thus, measuring similarity of words is relevant to software engineering research.

To measure similarity of words, the natural language processing community has created WordNet [20], [26]. WordNet is a general purpose lexical database that groups verbs, adjectives, nouns, and adverbs into cognitive synonym sets (aka. synsets). WordNet can be used to compute the semantic distance between two words [23]. However, due to the general purpose nature of WordNet, it does not contain many software-specific words. For example, words such as “src”, “cmd”, “WSDL”, “localhost”, etc. are not in WordNet. Furthermore, even when software-specific words do appear in WordNet, the semantic meaning stored in WordNet is often different. For example, WordNet relates “Eclipse” to words having to do with the moon, rather than to other integrated development environments (IDEs). Indeed, the study of Sridhara et al. showed that general English-based similarity measurements based on WordNet could not effectively be used to suggest similar words in software engineering context [31]. Thus, there is a need to build a specialized word similarity resource for the software engineering community.

There have been a number of initial efforts to build a word similarity resource specific for the software engineering community. Yang and Tan infer semantically related words in software source code [36]. Howard et al. mine similar verb pairs from comments and method signatures [11]. However, many software related words are not in the source code itself, but instead are in the various associated textual artifacts, such as forum posts, bug reports, commit logs, etc. Furthermore, many words used in a source code, especially abbreviations used as identifiers or used in method names, are specific to a particular project. Wang et al. infer semantically similar tags[1] in FreeCode [32]. However, they can only measure the similarity of tags and there are not many tags in FreeCode (690 in their experiments). In this work, we want to build a more generic word similarity resource that can be used for many different software engineering tasks, across a wide range of software projects.

Intuitively, two words are likely to be similar if they appear in similar contexts. For example, "client" and "tcp" often appear in sentences, paragraphs, or articles that describes networking. From this, we can infer that these two words are semantically related. Thus, we propose a new similarity metric called $WordSim^{SE}$ based on the concept of word co-occurrence [9] to measure the similarity of two words. We characterize each word using a co-occurrence vector that captures the co-occurrence of this word with popular software tags (primary anchors), other software tags (secondary anchors), and other words (tertiary anchors). We then compare each pair of words in terms of their co-occurrence vectors. Weights are used to quantify the contribution of each contextual word (i.e., a word that co-occurs with the target word) to the similarity of the word pairs.

We use our new similarity metric $WordSim^{SE}$ to construct $WordSim^{SE}_{DB}$, which is meant to be a specialized resource that mimics WordNet for software-specific terms. We take as input posts that appear in StackOverflow, a popular question-answering site. These posts contain many software-specific terms. We also leverage the phenomenon of tagging because it is very popular and it is supported by most software information sites including StackOverflow, FreeCode, SourceForge, etc. These tags are used to describe the major features of user generated contents and they are often important software-specific terms [35]. In this project, we use tags of posts on StackOverflow as *semantic anchors* to measure the similarity of two words.

We compare our approach, which builds $WordSim^{SE}_{DB}$, with an existing word similarity database computed by a WordNet-based word similarity approach [22], [23] (referred to as $WordNet^{res}$), using a set of software-specific words. For each such word, we use $WordSim^{SE}_{DB}$ and $WordNet^{res}$ to return the top-10 most similar words. We use ten human judges to evaluate the effectiveness of each approach by labeling the returned words as: related (score: 3), somewhat related (score: 2), and unrelated (score: 1), where "related" means that the two words are related *in the software engineering context*. We find that many software-specific words do not exist in $WordNet^{res}$. For those that exist in both $WordNet^{res}$ and $WordSim^{SE}_{DB}$, the average score of our approach is 2.31. This is 50.9% higher than the average score of the words returned using $WordNet^{res}$, which is 1.53.

Our contributions are as follows:

1) We build a software-specific word similarity resource leveraging 10,000 question-answer threads in StackOverflow.
2) We propose a novel similarity metric that leverages the phenomenon of tagging and characterizes a word in terms of a co-occurrence vector leveraging three kinds of semantic anchors: popular tags, other tags, and other words. Similarity of words is measured by computing the similarity of their corresponding representative vectors.
3) We have evaluated our approach on a number of software-specific words using ten human judges, who give scores on a 3-point Likert scale. Our experiment shows that our approach outperforms WordNet based word similarity significantly. Fifty-five percent of the software-specific words do not appear in WordNet. For those that appear in WordNet, the average Likert score of $WordNet^{res}$ is 1.53. On the other hand, $WordSim^{SE}$ can achieve an average Likert score of 2.31.

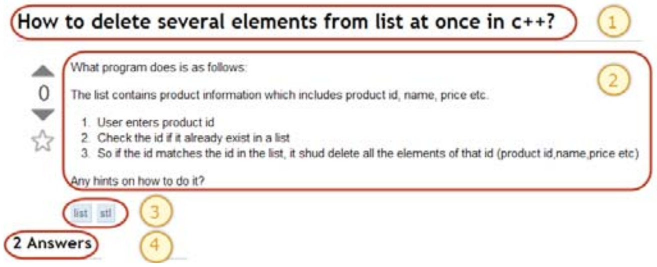
The structure of this paper is as follows. We introduce StackOverflow and several general text preprocessing steps in Section II. Our new similarity metric $WordSim^{SE}$ is defined in Section III. We present our proposed approach to construct $WordSim^{SE}_{DB}$ in Section IV. Our experiment methodology and results are presented in Section VI. We describe related work in Section VII. We finally conclude and discuss future work in Section VIII.

II. Preliminaries

In this section, we first introduce StackOverflow which is a popular question answering site. Next, we present several well-known text preprocessing steps: tokenization, stop-word removal, and stemming.

A. StackOverflow

StackOverflow[2] is one of the most popular question answering sites. It provides a platform for developers to help one another by asking and answering questions. With more than 1.7 million users and over 4,000,000 questions, StackOverflow has become large knowledge base. Most of the contents of StackOverflow are related to software development. In our work, we leverage posts in StackOverflow to construct a database that captures similarities between pairs of words.


How to delete several elements from list at once in c++?

What program does is as follows:

The list contains product information which includes product id, name, price etc.

1. User enters product id
2. Check the id if it already exist in a list
3. So if the id matches the id in the list, it shud delete all the elements of that id (product id,name,price etc)

Any hints on how to do it?

Fig. 1: A Sample Question and its Answers in StackOverflow

Figure 1 shows a question-and-answer thread extracted from StackOverflow. As illustrated by this example, each question in

[1] A tag is a short text (typically a word) that is used to label similar projects in FreeCode.

[2] stackoverflow.com

StackOverflow contains four parts: (1) a title, (2) a description of the question, (3) several tags, and (4) a set of zero or more answers. The title of the question is usually short. The description of the question provides more information that elaborates the title. The tags, which are provided by users of StackOverflow, are used to label the topics of the question and can be used to group related questions. Users can provide one or more answers to the question and these are recorded in the question-and-answer thread.

B. General Text Preprocessing

In the natural language processing domain, researchers apply several general text preprocessing steps to clean the textual dataset before analyzing it. This process includes: tokenization, stop-word removal, and stemming. We describe these steps as follows:

1) **Tokenization.** Tokenization is the process of breaking a paragraph (or a stream of characters) into its constituent word tokens. Delimiters are used to separate one word token from another. Punctuation marks and space are commonly used as delimiters.
2) **Stop-word Removal.** Some words, such as “the”, “is”, “and”, “of”, etc. carry little meaning, and are used in almost every document. These words are referred to as stop words. These stop words are typically removed from a textual dataset. In this work, we use a list of 641 stop words from an online stop word list [27].
3) **Stemming.** Stemming is the process of reducing a word to its root form. For example, both “reads” and “reading” can be reduced to “read”. In this paper, we use the well-known Porter stemming algorithm [25].

III. $WordSim^{SE}$: A Similarity Metric for Software-Specific Words

In this section, we describe how the similarity of two words is measured using $WordSim^{SE}$. Before we present $WordSim^{SE}$ in Section III-C, we first introduce the concepts of word co-occurrence and positive pointwise mutual information in Sections III-A & III-B, respectively.

A. Word Co-occurrence

The concept of word co-occurrence is defined based on the concept of “context”, which refers to the surrounding words of a target word [9]. A sliding window is used to limit the scope of the context of a target word. A target word should appear in the middle of the sliding window. For instance, a sliding window with size 5 would contain the target word itself, the two words appearing to the left of the target word, and the two words appearing to the right of the target word. If a target word appears in the beginning of a document, a sliding window with size 5 would only contain the target word and the two words appearing to the right of the target word. The co-occurrence frequency of a target word t with another word w is defined as the number of sliding windows of t (i.e., with t in the middle of the sliding window) containing w.

TABLE I: Example Document Corpus C_{Ex}

ID	Content	ID	Content
1	NetBeans Java	4	Eclipse Java
2	Java NetBeans	5	Java Eclipse
3	Eclipse Java	6	Eclipse Java

Example. Table I shows a corpus containing six documents. Consider a sliding window of size 3. The co-occurrence frequency of words NetBeans and Java is 2. This corresponds to two sliding windows: $\langle NetBeans, Java\rangle$ (Document 1), and $\langle Java, NetBeans\rangle$ (Document 2). The co-occurrence frequency of Eclipse and Java is 4. The co-occurrence frequency of NetBeans and Eclipse is 0.

B. Positive Pointwise Mutual Information ($PPMI$)

Various metrics have been proposed to measure the strength of associations (or relationships) between two words based on their co-occurrence frequencies. One popular metric is called pointwise mutual information (PMI) [8]. PMI comes from the information theory community and is used by the natural language processing community to measure the discrepancy between the *actual* co-occurrence frequency of two words and the *expected* co-occurrence frequency of the words assuming independence. If the PMI score of two words is larger than zero, it means the co-occurrences of the two words is not by chance and thus they are related. On the other hand, if the value is less than or equal to zero, then the two words are not related, even if they co-occur a large number times. Positive pointwise mutual information ($PPMI$) is a variation of PMI which returns the PMI score if the score is greater than zero, and zero otherwise. $PPMI$ has been reported to be the best co-occurrence based metric [3]. The definition of $PPMI$ is as follows:

$$P_d(i) = \frac{f_d(i)}{\sum_{i=1}^{n} f_d(i)} \tag{1}$$

$$P_d(i,j) = \frac{f_d(i,j)}{\sum_{i=1}^{n} f_d(i)} \tag{2}$$

$$PMI_d(i,j) = \log \frac{P_d(i,j)}{P_d(i)P_d(j)} \tag{3}$$

$$PPMI_d(i,j) = \begin{cases} PMI_d(i,j) & PMI_d(i,j) > 0 \\ 0 & otherwise \end{cases} \tag{4}$$

In the above equations, $P_d(i)$ is the probability of word i appearing in a document corpus d.[3] This probability can be estimated from $f_d(i)$ (i.e., the frequency of word i appearing in the corpus d) divided by the total number of words in the corpus. n stands for the number of unique words appearing in the corpus. $P_d(i,j)$ is the probability that two words i and j appear together in a sliding window in a document in the corpus. This probability can be estimated from $f_d(i,j)$ (i.e., the number of sliding windows containing words i and j in the corpus d) divided by the total number of words in the corpus. Note that the number of sliding windows is equal to the number of words in the corpus. $PMI_d(i,j)$ and $PPMI_d(i,j)$ are pointwise mutual information and positive

[3] A document corpus is simply a set of documents.

pointwise mutual information between word i and word j given a document corpus d. They can be computed based on $P_d(i)$, $P_d(j)$, and $P_d(i,j)$.

TABLE II: Frequency of Words in Document Corpus C_{Ex}

Variable	Value
f(NetBeans)	2
f(Eclipse)	4
f(Java)	6
f(NetBeans, Java)	2
f(Eclipse, Java)	4

Example. Table II shows the frequencies of various words and pairs of words of the corpus shown in Table I. The total number of words appearing in the corpus is 12. Based on the frequencies and the total number of words in the corpus we can compute $P_{C_{Ex}}(NetBeans)$, $P_{C_{Ex}}(Eclipse)$, $P_{C_{Ex}}(Java)$, $P_{C_{Ex}}(NetBeans, Java)$, and $P_{C_{Ex}}(Eclipse, Java)$, which are 0.17, 0.33, 0.5, 0.17, and 0.33 respectively. From these probabilities, $PMI(NetBeans, Java)$ and $PMI(Eclipse, Java)$ are both 1. Since these two values are larger than zero, the positive pointwise mutual information scores between NetBeans and Java and that between Eclipse and Java are both 1.

C. $WordSim^{SE}$

Next, we define our word similarity metric, $WordSim^{SE}$. To compute the similarity of two words, we represent them as vectors and then compute the similarity between these two vectors. Each word is represented as a feature vector where each element in the vector is the *co-occurrence weight* of that word with other (contextual) word in the corpus. These contextual words serve as *semantic anchors* forming a basis for us to compare the semantic distance of two words. The co-occurrence weight is measured using our weighted positive pointwise mutual information ($WPPMI$), defined by the following formula:

$$WPPMI_d(i,j) = W(j) \times PMI_d(i,j)$$

$$\text{where } W(j) = \begin{cases} \alpha & \text{(if } j \text{ is a popular software tag)} \\ \beta & \text{(if } j \text{ is a nonpopular software tag)} \\ \gamma & \text{(otherwise)} \end{cases} \tag{5}$$

In Equation 5, word i is the target word and word j is a contextual word. If word i does not co-occur with word j, the value of $WPPMI_d(i,j)$ is 0. $W(j)$ is a weight parameter used to control the contributions of different types of words serving as semantic anchors. We consider three kinds of words: popular software tags, other software tags, and other words. Software tags are typically software-specific terms and thus should be given higher weights (we treat them as *primary & secondary* semantic anchors). After the above weights are computed for each contextual word of the target word i, we have i's representative feature vector.

Next, to compute the similarity of two words, we compute the cosine similarity of their representative vectors. We denote this final similarity score as $WordSim^{SE}$, which is defined as:

$$WordSim^{SE}(i,j) = \frac{\sum_{k=1}^{n} WPPMI_d(i,k) \times WPPMI_d(j,k)}{\sqrt{\sum_{k=1}^{n} WPPMI_d(i,k)^2}\sqrt{\sum_{k=1}^{n} WPPMI_d(j,k)^2}} \tag{6}$$

In the above equation, n refers to the total number of unique words in the corpus. The rest of the notation used in Equation 6 is as defined for Equation 5.

Example. Consider the sample corpus in Table I. Let us assume that "Eclipse" and "Java" are popular software tags, while "NetBeans" is an unpopular software tag. Let the weights of popular tags, unpopular tags, and other words be 2, 1, and 0.5 respectively. Based on these, $WPPMI(NetBeans, Java)$ and $WPPMI(Eclipse, Java)$ are both 2. We now compute $WordSim^{SE}$(NetBeans, Eclipse). NetBeans and Eclipse both co-occur with "Java"; thus "Java" becomes the semantic anchor that connects the two words. "Java" is the *only* word that both NetBeans and Eclipse co-occur with. By taking the cosine similarity of the representative vectors of NetBeans and Eclipse, we have $WordSim^{SE}(NetBeans, Eclipse) = 1$.

IV. Construction Methodology

The previous section defines how the similarity of two words can be computed. In this section, we describe how we can use this information to automatically construct a database of similar word pairs. We first describe our overall process and then describe the key steps.

A. Overall Process

The overall process is shown in Figure 2. It contains four main steps: data preprocessing, word co-occurrence computation, parameter tuning, and similarity computation.

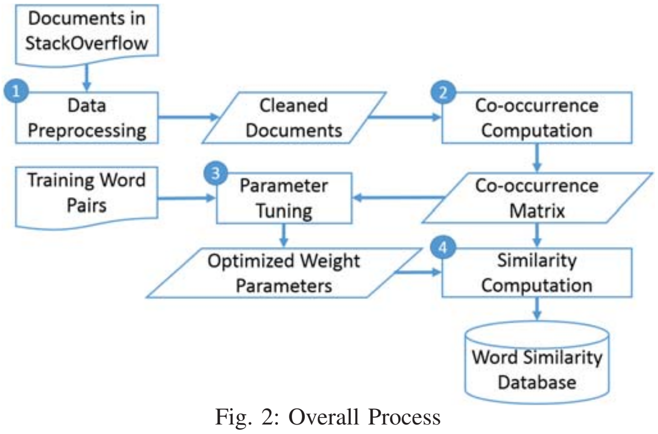


Fig. 2: Overall Process

Documents from StackOverflow are taken as the input of the data preprocessing step. Each document corresponds to a question and its corresponding answers in StackOverflow. The data preprocessing step cleans both the textual and code contents inside these documents and extracts tags from the documents. These tags are used to classify words into three kinds of semantic anchors: popular tags, other tags, and other words. The co-occurrence computation step counts the number of times each word pair co-occurs in the cleaned documents and generates a matrix to store the calculated numbers. Other information including the frequencies of words appearing in the document corpus and the total number of words in the corpus is recorded at the same time.

Next, the parameter tuning step optimizes the three weight parameters for the three types of semantic anchors, by taking the tags extracted in the data preprocessing step and the word co-occurrence matrix output by the co-occurrence computation step as inputs. The parameter tuning step performs this optimization by using a training set of similar word pairs. Note that the parameter tuning step depends on the co-occurrence matrix, and thus, for best results, it should be repeated whenever new data is added. Finally, the optimized weight parameters together with the co-occurrence matrix are sent to the similarity computation step, to calculate similarities between all pairs of word. We describe the details of these four main steps of our construction process below.

B. Step 1: Data Preprocessing

For each document extracted from StackOverflow, we apply the general text preprocessing steps: tokenization, stop-word removal, and stemming introduced in Section II-B to clean it. To clean the code segments inside a document, we have designed a code preprocessor to extract meaningful words. We apply code preprocessing before the general text processing. The code preprocessor cleans the code in the following ways:

1) **Removal of programming language keywords.** A programming language has its own keywords (reserved words) with special meaning that cannot be used as identifiers. In this work, the code preprocessor removes all Java programming language keywords from the documents.
2) **Splitting of identifiers.** The code preprocessor splits identifiers based on Camel casing and Pascal casing, which use capitalized characters to indicate the start of subsequent words in an identifier. The code preprocessor detects the capital letters inside identifier names and then splits each identifier into several words. Identifiers containing "_" are also broken into several words.

The data preprocessing component extracts tags from documents and ranks them in descending order based on the number of documents in which they occur. These tags and their usage frequencies are extracted to group words into three categories: popular tags, other tags, and other words. By default, the top-10% tags that are used in the most documents are treated as popular tags.

C. Step 2: Co-occurrence Computation

Following the definition of word co-occurrence in Section III, we first scan the cleaned documents to enumerate all the sliding windows. While scanning, we maintain and update a co-occurrence vector for each target word where features inside the vector correspond to the contextual words that co-occur with the target word. To save computation time and memory and to reduce noise, we follow the work of Xia et al. [35] and Wu et al. [33] by filtering words that appear less than a threshold, as they are rarely used (in this paper, by default, we remove words that appear fewer than 50 times in the corpus). We also delete word pairs whose co-occurrence frequencies are lower than a particular threshold – these word pairs are likely to co-occur by chance. By default, we delete word pairs whose co-occurrence frequencies are less than 5.

D. Step 3: Parameter Tuning

To tune the three weight parameters of $WordSim^{SE}$, i.e., α, β, and γ in Equation 5, we use a greedy search method. We use a manually labeled training set $WPair$ containing pairs of words that are most similar to each other. For each of the pairs, we take the first word in the pair, and use $WordSim^{SE}$ with one weight setting (α, β, and γ) to return a list of most similar words. We then investigate the rank of the second word of the pair in the returned list. The higher is the rank, the better it is. Based on this, the following fitness function is used to measure whether a weight setting can model well the contributions of the different types of words:

$$fitness(\alpha, \beta, \gamma) \quad = \quad \frac{1}{\sum_{(i,j)\in WPair} Rank_i(j)} \tag{7}$$

In the above equation, $Rank(j)$ is the rank of word j in the list of the most similar words to i computed based on $WordSim^{SE}$ with the particular weight setting under consideration. A low value of the rank (i.e., a larger number) suggests a worse weight setting. Thus, we would like to maximize this fitness function. The pseudocode of the parameter tuning process is shown in Figure 3.

```
Procedure TuneParameters
Inputs:
  $TagSet$ : A set of tag pairs used for tuning the weights.
  $min, max$: Minimum and maximum value of the weights.
  $\sigma$: A unit of weight change.
Output:
  Optimized $\alpha$, $\beta$, and $\gamma$ values.
Methods:
  // Step 1: Initialization
1 :   Let $\alpha = 3$, $\beta = 2$, and $\gamma = 1$
2 :   Let $fitVal^{best}$ = fitness($\alpha$, $\beta$, $\gamma$)
  // Step 2: Parameter Refinement
3 :   Do
4 :     Let $fitVal^{up}_{\alpha}$ = fitness($(\alpha + \sigma)$>$max?max$:$\alpha + \sigma$, $\beta$, $\gamma$)
5 :     Let $fitVal^{dn}_{\alpha}$ = fitness($(\alpha - \sigma)$<$min?min$:$\alpha - \sigma$, $\beta$, $\gamma$)
6 :     Let $fitVal^{up}_{\beta}$ = fitness($\alpha$, $(\beta + \sigma)$>$max?max$:$\beta + \sigma$, $\gamma$)
7 :     Let $fitVal^{dn}_{\beta}$ = fitness($\alpha$, $(\beta - \sigma)$<$min?min$:$\beta - \sigma$, $\gamma$)
8 :     Let $fitVal^{up}_{\gamma}$ = fitness($\alpha$, $\beta$, $(\gamma + \sigma)$>$max?max$:$\gamma + \sigma$)
9 :     Let $fitVal^{dn}_{\gamma}$ = fitness($\alpha$, $\beta$, $(\gamma - \sigma)$<$min?min$:$\gamma - \sigma$)
10:     Let $fitVal^{best}_{iter}$ = $Max_{i\in\{up,dn\},j\in\{\alpha,\beta,\gamma\}} fitVal^{i}_{j}$
11:     If $fitVal^{best}_{iter} > fitVal^{best}$
12:       $fitVal^{best}$ = $fitVal^{best}_{iter}$
13:       Update $\alpha$, $\beta$, or $\gamma$ accordingly
14:     Else
15:       Break
16:   While True
  // Step 3: Output
17: Output $\alpha$, $\beta$, $\gamma$
```

Fig. 3: Parameter Tuning Process

As shown in Figure 3, our algorithm consists of three steps. The first step (Lines 1-2) simply performs some initializations.

At line 1, the three weights α, β, and γ are initialized to 3, 2, and 1, based on the intuition that popular tags are more important than unpopular tags, which are more important than other words. Based on the fitness function defined in Equation 7, we compute a fitness score for this initial weight setting (Line 2). Next, we refine the parameters (i.e., weights) iteratively (Lines 3-16). For each iteration, we try to increment and decrement α, β, and γ by one unit of weight change σ, and compute new fitness scores using these weights (Lines 4-9). By default, we set this σ to be 0.1. If a weight (α, β, or γ) has reached the maximum (max) or the minimum (min) value, we do not increase or decrease it any further. We record the change in either α, β, or γ that results in the best fitness score (Line 10). If the best fitness score for this iteration is better than the best known fitness score, we update the best known fitness score and the weights accordingly (Lines 11-13). If the best score for an iteration is not better than the current best, we have reached a local optimum, and we terminate the refinement process (Lines 14-15). Finally, for the last step, we output the weights that result in the best fitness score.

E. Step 4: Similarity Computation

Once the optimized weight parameters α, β, and γ have been learned, we compute the similarities of the word pairs that appear in the corpus. Based on the definition of our weighted positive pointwise mutual information ($WPPMI$) and similarity metric $WordSim^{SE}$ defined in Equations 5 and 6, we construct a database that stores the similarities between all pairs of words.

The pseudocode for the similarity computation step is shown in Figure 4. We first initialize an empty word similarity database $WordSim^{SE}_{DB}$ (Line 1). Next, we scan the pre-processed words and enumerate all pairs of words (Lines 2-3). For each pair, we compute $WordSim^{SE}$ for the pair (Line 4). We store the similarity score into the database (Line 5). At the end, we output the database (Line 6).

```
Procedure ComputeWordSimilarity
Inputs:
    $Words^{all}$: The set of all pre-processed words in the corpus.
Output:
    $WordSim^{SE}_{DB}$: Word similarity database.
Methods:
1:  Initialize an empty $WordSim^{SE}_{DB}$
2:  For each word $w_i$ in $Words^{all}$
3:      For each word $w_j$ in $Words^{all}$
4:          Compute $WordSim^{SE}(w_i, w_j)$
5:          Add the computed similarity to $WordSim^{SE}_{DB}$
6:  Output $WordSim^{SE}_{DB}$
```

Fig. 4: Word Similarity Computation

V. Performance Optimization

The most time-consuming step of our construction process is the word similarity computation step (Step 4). In this step, all pairs of words are enumerated (Lines 2-3 of Figure 4). In this section, we propose an optimized approach. First, we describe the problems with the basic approach described in Section IV-E. Next, we present a new algorithm that addresses the shortcomings of the basic approach.

There are three main problems with the basic approach:

1) It is not necessary to compare each word with all other words. For a target word, the words that have no contextual words in common with it have zero similarity with the target word. Therefore, we can speed up the process by only considering words that have common contextual words with the target word. To address this problem, we use a set of inverted indices. Each inverted index corresponds to a contextual word c, and the index stores all words that co-occur with c. Given a target word t, these indices can be used to quickly find words that share at least one contextual word with t. Let $IdxSet$ be the set of indices and let $idx.W$ be the set of words pointed to by an index idx in $IdxSet$. The set of words that share at least one contextual word with t is then:

$$share\_context(t) = \bigcup_{idx \in IdxSet \wedge t \in idx.W} idx.W \setminus idx \tag{8}$$

2) There is no need to compute $WordSim^{SE}(j, i)$ if we have already computed $WordSim^{SE}(i, j)$, where i and j refers to any word appearing in the corpus. $WordSim^{SE}$ is defined as the cosine similarity of two vectors and therefore it is symmetric.

3) Based on the $WordSim^{SE}$ defined in Equation 6, we need to compute the length of word i's vector: $\sqrt{\sum_{k=1}^{n} WPPMI_d(i,k)^2}$, and perform a division operation multiple times during the enumeration of all pairs, which is a waste of time. To improve efficiency, we can change Equation 6 to the following equivalent formula:

$$WordSim^{SE}(i,j) = \sum_{k=1}^{n} Nm(WPPMI_d(i,k)) \times Nm(WPPMI_d(j,k)) \tag{9}$$

In the above formula, Nm($WPPMI_d(i,k)$) is the normalized $WPPMI$, defined as:

$$\frac{WPPMI_d(i,k)}{\sqrt{\sum_{k=1}^{n} WPPMI_d(i,k)^2}}$$

.

Using this formula, we can compute the normalized $WPPMI$ once and use it multiple times.

To address these three problems, we propose a new algorithm, which is shown in Figure 5. First, we initialize an empty word similarity database (Line 1). Then, an inverted index is created for each possible contextual word (Line 2). We also initialize a cache to store mappings between words and their corresponding normalized $WPPMI$ scores (Line 3). Then we iterate through each word i in the corpus (Lines 4-14). For each word i we compute its normalized $WPPMI$ scores and store them in the cache (if these scores have not been computed before) (Lines 5-7). This addresses the third problem. Then for word i we get the set of words $share\_context(i)$ computed by using the inverted indices based on Equation 8 (Line 8). These are words that share

```
Procedure ComputeWordSimilarity$^{Optimized}$
Inputs:
    $Words^{all}$: The set of all pre-processed words in the corpus.
Output:
    $WordSim_{DB}^{SE}$: Word similarity database.
Methods:
1 :  Initialize an empty $WordSim_{DB}^{SE}$
2 :  Generate inverted indices $L_1, L_2 ... L_n$ for all
         possible contextual words
3 :  Let Cache = Empty mapping from words to their
         normalized $WPPMI$ scores
4 :  For each word $i$ in $Words^{all}$
5 :    If $i$ is not in Cache
6 :      Compute normalized $WPPMI$ scores for i
7 :      Add the normalized scores to Cache
8 :    For each word $j$ in share_context(i)
9 :      If $j$ is not in Cache
10:        Compute normalized $WPPMI$ scores for j
11:        Add the normalized scores to Cache
12:      Compute $WordSim^{SE}(i, j)$ using Equation 9
13:      Add the computed similarity to $WordSim_{DB}^{SE}$
14:    Remove $i$ from the inverted indices
15:  Output $WordSim_{DB}^{SE}$
```

Fig. 5: Optimized Word Similarity Computation

at least one contextual word with i. This addresses the first problem. For each word j, we again compute its normalized $WPPMI$ score and store it in the cache (if the score has not been computed before) (Lines 9-11). For such a pair i and j, we compute word similarity using Equation 9 and add it to the database (Lines 12-13). After, we consider all js for word i, we delete i from the indices (Line 14). This step implies that we will not subsequently compute the similarity between word j and word i (which would be redundant). This addresses the second problem.

VI. Experiments & Analysis

In this section, we first describe the dataset used to construct $WordSim_{DB}^{SE}$ and the WordNet dataset that we use as a baseline. Next, we describe our experimental methodology and highlight our research questions. We answer these questions one by one. We also describe some threats to validity.

A. Dataset and Methodology

We construct $WordSim_{DB}^{SE}$ using the question-and-answer threads in StackOverflow. The data is provided by the MSR 2013 Mining Challenge [1]. The dataset is around 12 Gigabytes and contains all the posts[4] generated from August 2008-August 2012. In our work, we consider the questions and answers posted in January 2011. Since a question and its related answers are about the same topic, we organize the data into documents where each document contains a question and its corresponding answers. The title, description, and tags of the question and the contents of its answers are extracted and stored in the document. We have collected 83,358 documents. Out of them, we randomly sample 10,000 documents and use them to build $WordSim_{DB}^{SE}$. All the experiments are performed on an Intel Xeon X5460 3.16GHz server with 24.0GB RAM running Windows Server 2008 (64 bit).

For the third step of our construction methodology (see Section IV), we use 450 word pairs obtained from previous work to tune the weight parameters. We find that the best weights for α, β, and γ are 2.8, 2.0, and 1.4 respectively. This shows that indeed popular software tags are more important than other software tags, and that software tags are more important than other words.

We use the publicly available WordNet-based word pair similarity dataset in [24] as the baseline. This dataset contains the similarity of billions of word pairs, amounting to more than 100 Gigabytes of data. Word similarities are computed based on the Resnik metric [28]. In this work, we refer to this WordNet-based word similarity as $WordNet^{res}$.

We want to measure the effectiveness of $WordSim^{SE}$ and $WordNet^{res}$ in measuring the similarity of software-related words. To measure this, we follow these steps:

1) We take as input a set of N software-related words.
2) For each of the words in the set, we get the top-M words that are deemed to be most similar to it using $WordSim^{SE}$ and $WordNet^{res}$. We now have a set of $2 \times N \times M$ pairs of words.
3) We randomly mix these word pairs and assign them to P participants. Each word pair is assigned to one participant.
4) We ask the participants to label each pair of words using a 3-point Likert scale: 1 (Unrelated), 2 (Somewhat Related), and 3 (Related).
5) We now have $2 \times N$ lists of M Likert scores for $WordSim^{SE}$ and WordNet. We measure the average Likert scores of word pairs assigned by $WordSim^{SE}$ and those assigned by WordNet. We also measure a standard metric known as discounted cumulative gain (DCG) [17] which is given by the following formula:

$$DCG = \sum_{i=1}^{p} \frac{2^{rel_i} - 1}{\log_2(i+1)} \qquad (10)$$

In the above equation, rel_i refers to the Likert score of the word returned at rank i.

In our experiment, we set the numbers N, M, and P to be 100, 10, and 10 respectively. Ideally, 100 words can generate 2000 word pairs if both our approach and $WordNet^{res}$ are able to produce results. We manually select 100 meaningful software related words from questions on StackOverflow. [5] These words include:

> ant, css, cygwin, eclipse, gcc, git, soap, swing, ssl, svg, tcp, tomcat, trunk, wsdl, xcode, xhtml, xml, …[6]

The 10 participants of our user study include 9 graduate students and 1 post-doc from Singapore Management University. They work on research topics in the fields of Computer Science, and have at least 3 years of programming experience each in Java, C++, C#, etc.

[4] A post can be a question or an answer.

[5] These 100 words also appear in our sample data.

[6] The full list is available at: https://sites.google.com/site/wordsimdatabase/

B. Research Questions

We consider the following research questions:

RQ1 How accurate is our proposed approach as compared to the baseline approach?

RQ2 How general is our proposed approach in measuring the similarity of words?

RQ3 How scalable is our proposed approach?

From the information retrieval point of view, the first question measures precision while the second measures recall [17]. Precision is a measure of soundness while recall is a measure of completeness. The last question investigates the time it takes to construct $WordSim^{SE}_{DB}$ from a software corpus and the possibility to expand $WordSim^{SE}_{DB}$ by considering even more documents from StackOverflow and other software forums.

C. RQ1: Accuracy of $WordSim^{SE}$

Out of the 100 software-specific words, only 45 of them are in WordNet. In other words, the WordNet based approach can only return related words for 45 of the 100 selected words, while our approach can return related words for all 100. We perform a user study following the steps described in Section VI-A to judge whether the returned lists of words are really related to the target word. We describe the results using average Likert score and discounted cumulative gain (DCG) as yardsticks in the following paragraphs.

Average Likert Score. Table III shows the average Likert scores for our method and the baseline on the 45 words that exist in WordNet. The result shows that the words returned by our approach have an average Likert score of 2.31. The average Likert score of the words returned by $WordNet^{res}$ is only 1.53. Thus, we achieve a 50.9% improvement. The 2.31 average Likert score shows that $WordSim^{SE}$ can reasonably capture the semantic meaning of software-specific words. The average Likert score of our approach on the complete list of 100 words is 2.30.

TABLE III: Average Likert Score Comparison

Approach	Average Likert Score
$WordSim^{SE}$	2.31
$WordNet^{res}$	1.53
Improvement	50.9%

Average Discounted Cumulative Gain. The effectiveness of an information retrieval tool is impacted not only by the relevancy of the retrieved documents, but also by the ranking of the returned documents. More relevant documents should be ranked first. Thus in our setting, more similar words should be returned first. The DCG metric takes this into consideration. The higher the DCG, the better the algorithm performs. Table IV compares the average DCG for the 45 words on our approach and on the WordNet based approach. Table IV shows that our method outperforms the WordNet based approach by 66.9%. The average DCG of our approach across the 100 words is 25.99.

TABLE IV: Average Discounted Cumulative Gain Comparison

Approach	Average DCG
$WordSim^{SE}$	26.27
$WordNet^{res}$	15.74
Improvement	66.9%

D. RQ2: Generality of $WordSim^{SE}$

Table V shows the number of word pairs (with non-zero $WordSim^{SE}$ scores) in $WordSim^{SE}_{DB}$, the number of word pairs extracted in past studies, and the number of word pairs in WordNet. We manage to extract more than 5 million word pairs by analyzing 10,000 documents.

As shown in Table V, the number of word pairs in $WordSim^{SE}_{DB}$ is much smaller than that of WordNet. However, the number of word pairs extracted by our approach is nearly 12 times the number of word pairs extracted by Wang et al. from FreeCode tags [32]. It is also more than 4 times the number of word pairs extracted by Yang et al. from source code files of a number of large software projects [36].

TABLE V: Number of Word Pairs

Approach	#Word Pairs
$WordSim^{SE}_{DB}$	5,636,534
Wang et al. [32]	476,100
Yang et al. [36]	1,382,246
WordNet [24]	22,034,553,550

In this work, we have only evaluated the results of $WordSim^{SE}$ on 10,000 question-and-answer threads from StackOverflow. The number of word pairs can be increased by considering more question-and-answer threads from StackOverflow or other software forums. Thus, we believe our approach is more general than past proposed approaches that also extract software-specific similar word pairs [32], [36].

E. RQ3: Scalability of $WordSim^{SE}$

In this research question, we investigate the scalability of our proposed approach before and after performance tuning. We record the running time to process 10,000, 20,000, 30,000, 40,000, and 50,000 documents. Figure 6 shows the number of word pairs in each document corpus.

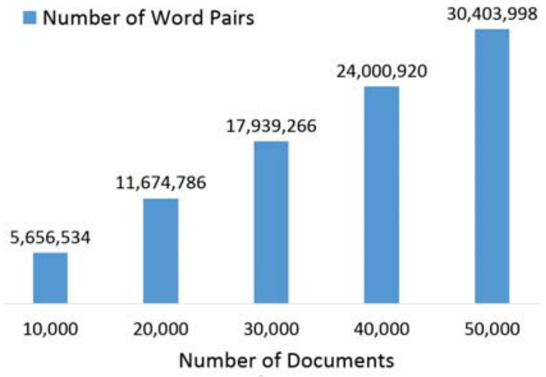


Fig. 6: Number of Word Pairs in Each Document Corpus

We plot the runtime as a function of the number of documents and show the result in Figure 7. Note that the runtime includes the time for data preprocessing, co-occurrence computation, parameter tuning, and similarity computation. The result shows that the optimized approach (described in Figure 5) is around 4-6 times faster than the basic approach (described in Figure 4). We also notice that the runtime grows more or less linearly with the number of documents used to construct $WordSim_{DB}^{SE}$. To construct the database using 50,000 documents, using the optimized approach, we need less than 15 minutes.

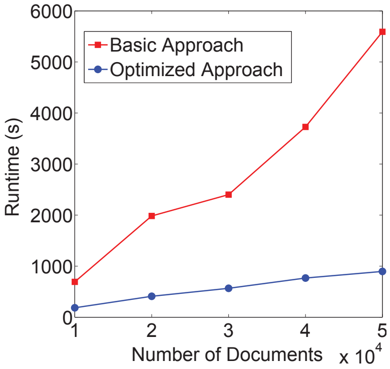


Fig. 7: Runtime Comparison: Before and After Performance Tuning

F. *Threats to Validity*

Threats to internal validity relate to experimental errors and biases. When performing the user study, to reduce bias we do not tell the participants which tool generated the word pairs. Otherwise participants might be inclined to give a higher score to pairs generated by our approach than those generated by the baseline approach. We follow many previous studies [4], [11], [32] by measuring the effectiveness of the approaches by using human labeled Likert scores. Since performing a user study is expensive, we give each word pair to only one participant. This might cause a subjectivity bias and implies that we could not compute inter-rater reliability. Many past studies [4], [11], [32] also do not compute inter-rater reliability, mainly due to the cost of performing a user study on a large number of participants and a large number of tasks. Another threat is the fact that the participants of our user study are graduate students and a post-doc, rather than experts from industry. However, it has been reported that only minor differences exist between the performance of graduate students and professional software developers when performing relatively small tasks of judgement [10].

Threats to external validity relate to the generalizability of our approach. We consider 10,000 question-and-answer threads from StackOverflow. We use 100 software-specific words to evaluate our approach. In the future, we would like to reduce this threat of validity further by analyzing more threads from StackOverflow and similar sites. We would also like to investigate the effectiveness of our approach using more software-specific words and more participants.

Threats to construct validity relate to the suitability of our evaluation metrics. We use two metrics: average Likert score and discounted cumulative gain (DCG). The first metric is very intuitive, and the second is also well known. Thus, we believe we have little threat to construct validity.

VII. Related Work

A. *Generic Word Similarity Measurements*

Measuring similarity between two words is one of the basic natural language processing (NLP) tasks. Many papers have proposed various approaches to measure this similarity. Most of the well-known techniques leverage a lexical database (e.g., WordNet) to measure similarities of words [14], [15], [28], [34]. Pedersen et al. have created an interface to allow users to query the semantic distance of words [23]. They have also pre-computed the similarities of all pairs of words in WordNet and make these similarities publicly available [24]. Similar to the above studies, we also measure similarities of words. However, rather than trying to leverage WordNet which is a general purpose resource, we build a WordNet-like resource specific to the software engineering domain that measures similarities of words in the software engineering context.

A number of approaches have also been proposed to automatically construct a thesaurus [6], [15]. They are based on the distributional hypothesis, which assumes that words that co-occur in the same contexts are likely to have similar meanings. Other co-occurrence based similarity measurements can be found in [2], [13]. We also make use of word co-occurrence to measure similar words in software engineering context. Our work is different from theirs in two aspects: we leverage the phenomenon of tagging in the software engineering community, and we make use of a dataset unique to the software engineering domain rather than a general dataset. Tags in StackOverflow are typically software-specific words and we use these as the primary and secondary semantic anchors to measure the similarity between words.

B. *Software-Specific Word Similarity*

Yang and Tan infer semantically related words in software source code files [36]. Their approach takes as input a code base along with a list of stop words and produces a set of semantically related pairs of words. It was able to identify semantically related words with a reasonable accuracy in 9 large and popular code bases written in C and Java. Similar with the work of Yang and Tan, Howard et al. infer semantically similar verbs from comments and method signatures [11]. They mined 97 similar verb pairs from 150 methods which are randomly sampled from 36 Java programs across multiple domains. In this work, we also generate semantically related words. However, rather than analyzing code, we analyze software-specific textual documents (namely posts in StackOverflow). We also propose a different algorithm to analyze the different dataset. Many software related words are not in source code but in the various software textual artifacts, e.g., forum posts, bug reports, etc.

Wang et al. infer semantically related tags from FreeCode, a website that maintains a large index of software [32]. Each software project on FreeCode has a document to describe its features. Tags are used to label similar projects. Wang et al. measure similarity of tags by considering document similarity and textual similarity. We also measure similarity of words and leverage software engineering contents. However, our approach is more general than that of Wang et al., which is only able to measure similarity between tags. There are not so many tags in FreeCode; in their experiments, they only measure the similarities of 690 tags. Our approach, in contrast, measures the similarity of words that appear in many posts.

C. Other Related Studies

Falleri et al. extract a network of identifiers connected by "is-more-general-than" or "is-a-part-of" relationships from source code [7]. In this work, we are not interested in building a network of terms connected by the two relationships, rather we want to measure the similarities of words considering the software engineering context.

VIII. Conclusion and Future Work

Many automated software engineering tools analyze software artifacts leveraging natural language processing (NLP) tools. Measuring word similarity is one of the basic NLP tasks and has been shown to improve various NLP tools. The natural language processing community has released WordNet, a lexical database, which has been leveraged to measure similarity of words. However, WordNet is general purpose and often does not contain software-specific terms. Furthermore, the meanings of words in WordNet are often different than when they are used in software engineering context. Thus, there is a need for a domain-specific WordNet-like resource for the software engineering community.

In this work, we propose a technique that automatically constructs a database called $WordSim_{DB}^{SE}$ that stores the similarity of words used in software engineering context. We proposed a similarity metric $WordSim^{SE}$ leveraging question-and-answer threads in StackOverflow to infer the similarity of words based on their weighted co-occurrences with three kinds of semantic anchors. We have compared our approach with a WordNet-based approach ($WordNet^{res}$) by means of a user study. Our user study shows that our approach outperforms $WordNet^{res}$ in terms of average Likert score and average discounted cumulative gain (DCG) by more than 50% and 66%, respectively. We have optimized the steps to construct $WordSim_{DB}^{SE}$. The result of the scalability test demonstrates that our approach is around 4-6 times faster than the basic approach. Our optimized approach can estimate the similarity of more than 30 million word pairs in less than 15 minutes by analyzing a 50,000 document corpus.

In the future, we plan to construct a larger $WordSim_{DB}^{SE}$ by training it with more question-and-answer threads from StackOverflow and other resources that contain software related documents. In fact, our scalability experiment already implies that the runtime would not drastically increase if we include more threads from StackOverflow. We also plan to allow open access to an expanded $WordSim_{DB}^{SE}$ as a web service.

References

[1] A. Bacchelli, "Mining challenge 2013: Stack overflow," in *MSR*, 2013.

[2] D. Bollegala, Y. Matsuo, and M. Ishizuka, "Measuring semantic similarity between words using web search engines," *WWW*, 2007.

[3] J. Bullinaria and J. Levy, "Extracting semantic representations from word co-occurrence statistics: A computational study," *Behavior Research Methods*, 2007.

[4] R. P. Buse and W. R. Weimer, "Learning a metric for code readability," *TSE*, vol. 36, no. 4, 2010.

[5] H. Cao, D. Jiang, J. Pei, Q. He, Z. Liao, E. Chen, and H. Li, "Context-aware query suggestion by mining click-through and session data," in *KDD*, 2008.

[6] L. Chen, P. Fankhauser, U. Thiel, and T. Kamps, "Statistical relationship determination in automatic thesaurus construction," in *CIKM*, 2005.

[7] J.-R. Falleri, M. Huchard, M. Lafourcade, C. Nebut, V. Prince, and M. Dao, "Automatic extraction of a wordnet-like identifier network from software," in *ICPC*, 2010.

[8] R. Fano, *Transmission of Information: A Statistical Theory of Communications*. MIT Press, 1961.

[9] Z. Harris, *Mathematical Structures of Language*. New York, USA: Wiley, 1968.

[10] M. Höst, B. Regnell, and C. Wohlin, "Using students as subjects: a comparative study of students and professionals in lead-time impact assessment," *Empirical Software Engineering*, vol. 5, no. 3, 2000.

[11] M. J. Howard, S. Gupta, L. Pollock, and K. Vijay-Shanker, "Automatically mining software-based, semantically-similar words from comment-code mappings," in *MSR*, 2013.

[12] S. F. Hussain and G. Bisson, "Text categorization using word similarities based on higher order co-occurrences," in *SDM*, 2010.

[13] A. Islam and D. Inkpen, "Second order co-occurrence PMI for determining the semantic similarity of words," in *LREC*, 2006.

[14] J. J. Jiang and D. W. Conrath, "Semantic similarity based on corpus statistics and lexical taxonomy," *ROCLING X*, 1998.

[15] D. Lin, "An information-theoretic definition of similarity," in *ICML*, 1998.

[16] E. Linstead, S. K. Bajracharya, T. C. Ngo, P. Rigor, C. V. Lopes, and P. Baldi, "Sourcerer: mining and searching internet-scale software repositories," *Data Min. Knowl. Discov.*, vol. 18, no. 2, 2009.

[17] C. Manning, P. Raghavan, and H. Schutze, *Introduction to information retrieval*. Cambridge University Press Cambridge, 2008, vol. 1.

[18] A. Marcus, A. Sergeyev, V. Rajlich, and J. I. Maletic, "An information retrieval approach to concept location in source code," in *WCRE*, 2004.

[19] T. Menzies and A. Marcus, "Automated severity assessment of software defect reports," in *ICSM*, 2008.

[20] G. Miller, "Wordnet: a lexical database for English," *Communications of the ACM*, 1995.

[21] T. Pedersen, "Information content measures of semantic similarity perform better without sense-tagged text," in *HLT-NAACL*, 2010.

[22] T. Pedersen, S. Patwardhan, and J. Michelizzi, "Wordnet: : Similarity - measuring the relatedness of concepts," in *AAAI*, 2004.

[23] T. Pederson, "Wordnet::similarity," http://wn-similarity.sourceforge.net/.

[24] M. Porter, "An algorithm for suffix stripping," *Program*, vol. 14, 1980.

[25] Princeton University, "WordNet: A lexical database for English," http://wordnet.princeton.edu/.

[26] Ranks.NL, "English stopwords," http://www.ranks.nl/resources/stopwords.html.

[27] P. Resnik, "Using information content to evaluate semantic similarity in a taxonomy," in *IJCAI*, 1995.

[28] M. Roldan-Vega, G. Mallet, E. Hill, and J. A. Fails, "Conquer: A tool for NL-based query refinement & contextualizing code search results," in *ICSM*, 2013.

[29] P. Runeson, M. Alexandersson, and O. Nyholm, "Detection of duplicate defect reports using natural language processing," in *ICSE*, 2007.

[30] G. Sridhara, E. Hill, L. Pollock, and K. Vijay-Shanker, "Identifying word relations in software: A comparative study of semantic similarity tools," in *ICPC*, 2008.

[31] S. Wang, D. Lo, and L. Jiang, "Inferring semantically related software terms and their taxonomy by leveraging collaborative tagging," in *ICSM*, 2012.

[32] L. Wu, L. Yang, N. Yu, and X.-S. Hua, "Learning to tag," in *WWW*, 2009.

[33] Z. Wu and M. Palmer, "Verbs semantics and lexical selection," in *ACL*, 1994.

[34] X. Xia, D. Lo, X. Wang, and B. Zhou, "Tag recommendation in software information sites," in *MSR*, 2013.

[35] J. Yang and L. Tan, "Swordnet: Inferring semantically related words from software context," *Empirical Software Engineering*, 2013.