2017 IEEE/ACM 39th International Conference on Software Engineering

# Unsupervised Software-Specific Morphological Forms Inference from Informal Discussions

Chunyang Chen*, Zhenchang Xing† and Ximing Wang*
*School of Computer Science and Engineering, Nanyang Technological University, Singapore
chen0966@e.ntu.edu.sg; wang0893@e.ntu.edu.sg;
†Research School of Computer Science, Australian National University, Australia
zhenchang.xing@anu.edu.au

***Abstract*—Informal discussions on social platforms (e.g., Stack Overflow) accumulates a large body of programming knowledge in natural language text. Natural language process (NLP) techniques can be exploited to harvest this knowledge base for software engineering tasks. To make an effective use of NLP techniques, consistent vocabulary is essential. Unfortunately, the same concepts are often intentionally or accidentally mentioned in many different morphological forms in informal discussions, such as abbreviations, synonyms and misspellings. Existing techniques to deal with such morphological forms are either designed for general English or predominantly rely on domain-specific lexical rules. A thesaurus of software-specific terms and commonly-used morphological forms is desirable for normalizing software engineering text, but very difficult to build manually. In this work, we propose an automatic approach to build such a thesaurus. Our approach identifies software-specific terms by contrasting software-specific and general corpuses, and infers morphological forms of software-specific terms by combining distributed word semantics, domain-specific lexical rules and transformations, and graph analysis of morphological relations. We evaluate the coverage and accuracy of the resulting thesaurus against community-curated lists of software-specific terms, abbreviations and synonyms. We also manually examine the correctness of the identified abbreviations and synonyms in our thesaurus. We demonstrate the usefulness of our thesaurus in a case study of normalizing questions from Stack Overflow and CodeProject.**

## I. Introduction

Informal discussions on social platforms (such as Stack Overflow, CodeProject) have become a common means for developers to share and acquire programming knowledge in natural language text. Many natural-language-processing (NLP) based techniques have been proposed to mine programming knowledge from such informal discussions to assist software engineering tasks, such as document search [59], [60], extracting API mentions and usage insights [58], [53], recovering traceability between informal discussions (e.g., duplicate question [55]) or between code and informal discussions [42], linking domain-specific entities in informal discussions to official documents [50] and mining technology landscape [12], [15]. To make an effective use of NLP techniques in these tasks, a consistently-used vocabulary of software-specific terms is essential, because NLP techniques assume that the same words are used whenever a particular concept is mentioned.

TABLE I: Morphological forms of *visual c++*

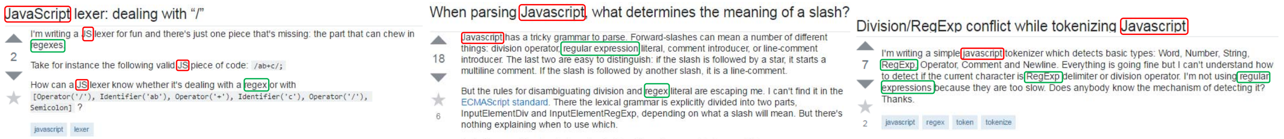
| Term | Frequency | Annotation |
|---|---|---|
| visual c++ | 10,294 | Standard |
| msvc | 8,477 | abbreviation |
| vc++ | 7,154 | abbreviation |
| microsoft visual c++ | 1,826 | synonym |
| ms vc++ | 295 | abbreviation |
| visual-c++ | 110 | synonym |

As informal discussions are contributed by millions of users with very diverse technical and linguistic background, the same concept is often mentioned in many *morphological forms, including abbreviations, synonyms and misspellings*, intentionally or accidentally [58]. Fig. 1 shows three Stack Overflow posts that discuss the slash issue of regular expression when parsing JavaScript. These three posts are marked as duplicate posts by the Stack Overflow community, because they discuss the same programming issue. That is, the three posts are considered as semantically equivalent. However, when mentioning regular expression and JavaScript, the three different users use many morphological forms (e.g., *regex*, *RegExp*, *regexes*), and even the same user uses various forms in the same post (e.g., *JS*, *JavaScript*). As another example, Table I summarizes the frequencies of various morphological forms of *visual c++* in Stack Overflow discussions. Note that there are many morphological forms for the same concept and some forms are used as frequently as the standard one.

The wide presence of morphological forms of the same concept in informal discussions poses a serious challenge to informal retrieval. For example, for the query "slash in regular expressions Javascript", some posts in Fig. 1 may not be retrieved due to the morphological forms of *JavaScript* and *regular expression* used in the posts, even though the three ports are semantically equivalent. Overlooking the close relationships between various morphological forms of the same concept may also accentuate data sparsity problems in applying NLP techniques to mining programming knowledge in informal discussions, which could negatively affect the performance of the NLP techniques.

In the NLP domain, a recent trend has seen proposals that deal with morphology using word embeddings and neural networks [17], [10], [34]. A recent work by Soricut and Och [48] exploits the relational regularities exhibited by word embeddings (e.g., car to cars, dog to dogs) to model prefix- and suffix-based morphological rules and transformations. However, these morphology learning techniques in the NLP domain consider only morphological relations drawn out of

1558-1225/17 $31.00 © 2017 IEEE
DOI 10.1109/ICSE.2017.48



JavaScript lexer: dealing with “/”

I'm writing a JS lexer for fun and there's just one piece that's missing: the part that can chew in regexes.

Take for instance the following valid JS piece of code: `/ab+c/;`

How can a JS lexer know whether it's dealing with a regex or with `[Operator('/'), Identifier('ab'), Operator('+'), Identifier('c'), Operator('/'), Semicolon]` ?

(a) post ID=14553203

When parsing Javascript, what determines the meaning of a slash?

Javascript has a tricky grammar to parse. Forward-slashes can mean a number of different things: division operator, regular expression literal, comment introducer, or line-comment introducer. The last two are easy to distinguish: if the slash is followed by a star, it starts a multiline comment. If the slash is followed by another slash, it is a line-comment.

But the rules for disambiguating division and regex literal are escaping me. I can't find it in the ECMAScript standard. There the lexical grammar is explicitly divided into two parts, InputElementDiv and InputElementRegExp, depending on what a slash will mean. But there's nothing explaining when to use which.

(b) post ID=5519596

Division/RegExp conflict while tokenizing Javascript

I'm writing a simple javascript tokenizer which detects basic types: Word, Number, String, RegExp, Operator, Comment and Newline. Everything is going fine but I can't understand how to detect if the current character is RegExp delimiter or division operator. I'm not using regular expressions because they are too slow. Does anybody know the mechanism of detecting it? Thanks.

(c) post ID=4726295

Fig. 1: The morphological forms of “regular expression” (blue) and “javascript” (red) in three Stack Overflow posts

linguistic regularities of natural language (e.g., prefix ans suffix). As shown in Fig. 1 and Table I, morphological forms of software-specific terms found in informal discussions do not always follow linguistic regularities of natural language, e.g., *JS and Javascript*, *RegExp and regular expressions*.

In the software engineering domain, the impact of consistent vocabulary on the application of NLP-based techniques to source code and software documentation has long been recognized [30], [29], [19], [28], [23]. The focus has been on expanding identifiers that contain abbreviations and acronyms. The proposed solutions are predominantly lexically-based approaches, for example, based on common naming conventions in software engineering like camel case, or use string edit distance to measure the similarity between an abbreviation and its potential expansions. But lexical rules are often unreliable. For example, both *open cv* and *opencsv* are lexically similar to *opencv*. However, *opencsv* is a library for parsing *csv* files which is totally irrelevant to *opencv* (a computer vision library). To improve the results, most of these approaches resort to external resources (e.g., English dictionary, dictionary of IT term and known abbreviations) which are often difficult to build and maintain, especially domain-specific ones. Some approaches [51], [56], [26] exploit word frequencies in word co-occurrence data to rank abbreviation expansions, but none of them exploit semantic relatedness of words.

In this paper, we propose an automatic approach for inferring morphological forms of software-specific terms in a large corpus of informal software engineering text. Our approach first contrasts software engineering text (e.g., Stack Overflow discussions) against general text (e.g., Wikipedia documents) to derive a vocabulary of software-specific terms used in software engineering text. It then combines the latest development of word embeddings in the NLP domain and the domain-specific lexical rules developed in the software engineering domain. As such, we can infer morphological forms of software-specific terms that not only obey lexical rules but also are semantically close to each other. Based on the graph of the morphological relations between pairs of terms, our approach find groups of morphological forms, each expressing a distinct concept (see Fig. 5 for examples), similar to the notion of synset of WordNet [37].

Compared with several community-curated lists of IT terms, abbreviations and synonyms, our approach automatically infers software-specific terms and morphological forms that are up-to-date and actively used in Stack Overflow. In contrast, community-curated lists contain many out-of-date terms and rarely-used morphological forms. This result demonstrates the needs for and the advantage of an approach like ours for automatic software-specific morphological form inference. Manual examination of randomly sampled 1,200 abbreviations and synonyms confirms the high accuracy (81.9%) of our approach. To demonstrate the usefulness of the inferred morphological forms of software-specific terms for information retrieval, we use the inferred morphological forms to normalize question text and question metadata (i.e., tags) from both Stack Overflow and CodeProject. Our results show that our morphological forms can better improve the consistency between question text and question metadata, compared with Porter stemming [45] and WordNet-based lemmatization [7] that are commonly used for English text normalization. Furthermore, the results show the generality of our morphological forms across different software engineering corpus.

## II. Related Work

When developing software, developers often use abbreviations and acronyms in identifiers and domain-specific terms that must be typed in source code and documentation. This phenomena challenges the effectiveness of NLP-based techniques in exploiting software text. Previous attempts at expanding abbreviations in identifiers have looked at string edit distance [16], string frequencies from source code [38], [20], word co-occurrence data [28], or a combination of several techniques. These approaches inspire the design of lexical rules and transformations in our approach.

Informal discussions on social platforms (e.g., Stack Overflow) contains many abbreviations, synonyms and misspellings. Furthermore, informal discussions on social platforms cover a much broader range of programming knowledge, compared with traditional software text that earlier work has focused on. These facts could render heuristics developed for expanding abbreviated identifiers unreliable when applied to informal discussions. For example, *jsp* may be regarded as the abbreviation of *javascript*, *google apps* as the synonym of *google apis*, or *yaml* as the synonym of *xaml*.

To reliably infer morphological forms of software-specific terms in a broad range of informal discussions, we must analyze semantic relatedness of software-specific terms and morphological forms. One way is to develop a software-specific thesaurus like the WordNet [37] for general text, and the BioThesaurus [33] for biology domain. In fact, most of identifier expansion approaches [30], [29], [19], [28], [23] use in a way or the other external dictionary, such as general English dictionary, domain-specific ontology and/or a list of well-known abbreviations to rank the relatedness of abbreviation and expansions. Although such dictionary is useful for reasoning word relatedness, it requires significant time and efforts to build and it is difficult to scale it up and keep it up-to-date. This is why we do not want our approach to rely on such dictionary. Indeed, our work presents an automatic approach to mine a high-quality thesaurus of software-specific terms and morphological forms by analyzing the text data alone.

Several attempts have been made to automatically infer semantically related terms in software engineering text [46], [49], [28], [26], [51], [56]. However, the proposed techniques make some important assumptions about the text, for example, relying on project-specific naming or documentation convention [47], [26], or lexical difference patterns between sentences [56], or the availability of certain contextual information (e.g., dictionary, question tags) [28], [52], [54]. Such assumptions lack the generality for other data. Unlike existing approaches, we resort to unsupervised word representations (i.e., word embeddings) [36] to capture word semantics from a large corpus of unlabeled software engineering text. Recently, word embeddings have been successfully applied to various software engineering problems involving text data, such as document retrieval [59], [24], software-specific entity recognition [58], analogical library recommendation [11], [13], prediction of semantically related posts [55]. Our goal is different: based on semantically similar software-specific terms, we infer a group of morphological forms which represent a distinct concept in informal discussions.

Beyer and Pinzger [8] develop a tag synonym suggestion tool to generate tag synonyms. Their method is based on rules derived from the human observations of existing tag synonym pairs in Stack Overflow. Their follow-up work [9] uses community detection technique to group tag synonym pairs into topics, which can assist in the analysis of topic trend on Stack Overflow. Our approach differs from their work in two aspects: 1) their work involves only about 3,000 known tag synonym pairs, while our work identifies over 52,000 software-specific terms and discovers morphological relations among these terms; 2) their work studies only Stack Overflow tags, while our work applies the morphological forms mined from Stack Overflow to normalize the text from CodeProject.

## III. THE APPROACH

As shown in Fig. 2, the input to our approach is only a software-specific corpus (e.g., Stack Overflow text) and a general corpus (e.g., Wikipedia text). Our approach includes six main steps: 1) text cleaning and phrase detection, 2) identifying software-specific vocabulary by contrasting software-specific and general corpus, 3) learning term semantics by word embedding techniques (e.g., continuous skip-gram model [36]), 4) extracting semantically related terms as candidates of morphological forms, 5) discriminating abbreviations and synonyms from the list of morphological-form candidates, and 6) based on a graph of morphological relations, grouping morphological forms of software-specific terms. The output of our approach is a thesaurus of software-specific terms and their morphological forms (called $SEthesaurus$).

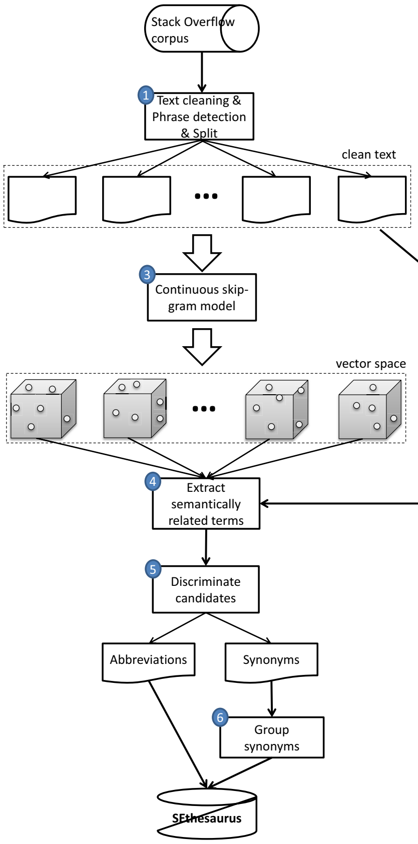

Stack Overflow corpus
Wikipedia corpus
1
Text cleaning & Phrase detection & Split
1
Text cleaning & Phrase detection
clean text
•••
Clean text
3
Continuous skip-gram model
2
Compare
vector space
•••
Software-specific vocabulary
4
Extract semantically related terms
5
Discriminate candidates
Abbreviations
Synonyms
6
Group synonyms
SEthesaurus

Fig. 2: The overview of our approach

### A. Dataset

Our approach takes a software-specific corpus of plain text and a general corpus of plain text as inputs. No other external resources are required. Software specific corpus can be crawled from domain-specific websites, such as Stack Overflow, CodeProject, W3School, MSDN. As we are interested in discovering morphological forms in informal discussions, as well as considering the popularity of the website and the volume of the data, we choose Stack Overflow text as software-specific corpus in this work. General corpus can be crawled from domain-agnostic websites, such as Wikipedia, Quora, Baidu Zhidao, which cover a diverse set of domains. Considering the quality and the public availability of the data, we choose Wikipedia text as general corpus in this work. Wikipedia text is also adopted as general corpus in other NLP work [39]. It is important to note that our data-analysis approach is not limited to Stack Overflow and Wikipedia data.

**Raw dataset:** In this work, the Stack Overflow data dump [4] we use contains 9,970,064 questions and 16,502,856 answers from July 2008 to August 2015. We collect the title and body content of all the questions and answers as the software-specific corpus. The Wikipedia data dump [6] includes 5,044,130 articles before December 2015. We collect the page content of all the articles as the general corpus.

### *B. Preprocessing Input Corpuses*

*1) Text cleaning:* As both datasets are from websites, we follow the text cleaning steps commonly-used for preprocessing web content [44], [43]. We preserve textual content but remove HTML tags. For Wikipedia data, we remove all references from page content. For Stack Overflow, we remove long code snippets in <pre><code> in the posts, but not short code elements in <code> in natural language sentences. We use our software-specific tokenzier [57] to tokenize the sentences. This tokenizer preserves the integrity of code-like tokens and the sentence structure. For example, it treats $pandas.DataFrame.apply()$ as a single token, instead of a sequence of 7 tokens, i.e., pandas . DataFrame . apply ( ).

*2) Phrase Detection:* A significant limitation of priori techniques is that they consider only single word. However, many software engineering terms are composed of several words such as *ruby on rails*, *visual studio* and *depth first search*. These multi-words phrases must be recognized and treated as a whole in data analysis.

We adopt a simple data-driven and memory-efficient approach [36] to detect multi-words phrases in the text. In this approach, phrases are formed iteratively based on the unigram and bigram counts, using

$$score(w_i, w_{i+1}) = \frac{count(w_i w_{i+1}) - \delta}{count(w_i) \times count(w_{i+1})} \quad (1)$$

The $w_i$ and $w_{i+1}$ are two consecutive words. $\delta$ is a discounting coefficient to prevent phrases consisting of two infrequent words to be formed. That is, the two consecutive words will not form a bigram phrase if they appear as a phrase less than $\delta$ times in the corpus. In this work, we experimentally set $\delta$ at 10 and the threshold for $score$ at 15 to achieve a good balance between the coverage and accuracy of the detected multi-words phrases.

Our method can find bigram phrases that appear frequently enough in the text compared with the frequency of each unigram, such as *sql server*. But the bigram phrases like *this is* will not be formed because each unigram also appear very frequently separately in the text. Once the bigram phrases are formed, we repeat the process to detect trigram and fourgram phrases. In this work, we stop at fourgram phrases, but the approach can be extended to longer phrases.

**Corpus summary:** After text cleaning and phrase detection, we obtain a software-specific corpus from the Stack Overflow data dump, which has 8,125,944 unique terms (including single words and multi-words phrases) and 1,757,436,186 tokens (a token is a mention of a term). We obtain a general corpus from the Wikipedia data dump, which has 26,639,445 unique terms and 2,356,736,103 tokens.

### *C. Building Software-Specific Vocabulary*

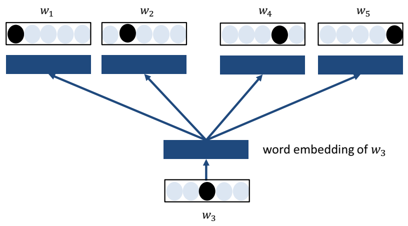


Fig. 3: The continuous skip-gram model. It predicts the surrounding words given the center word.

Inspired by Park et al's work [39], we identify software-specific terms by contrasting the term frequency of a term in the software specific corpus compared with its frequency in the general corpus. Specially, we measure the a term's *domain specificity* based on the equation:

$$domainspecificity(t) = \frac{p_d(t)}{p_g(t)} = \frac{\frac{c_d(t)}{N_d}}{\frac{c_g(t)}{N_g}} \quad (2)$$

where $d$ and $g$ represents software-specific and general corpus respectively, and $p_d(t)$ and $p_g(t)$ is the probability of the term $t$ in the two corpuses respectively. The probability of a term in a corpus is calculated by dividing the term frequency by the total number of tokens $N$ in the corpus. The underlying intuition is that terms that appear frequently in software-specific corpus but infrequently in general corpus are software-specific terms. In this work, we experimentally set 10 as the threshold for $domainspecificity$ to discriminate software-specific terms.

We observe that some terms that developers commonly use on Stack Overflow bear little domain-specific meaning. For example, *i* is frequently used as variable in loop. Developers also frequently mention some numeric metrics, such as *1 sec* and *10mb*. As these terms do not represent any domain-specific concepts in natural language discussions, we set stop-term rules to exclude such meaningless terms, for example, excluding terms beginning with number or special punctuations like *, + and >, excluding terms with only one letter (*c* and *r* are preserved as they are programming languages).

### *D. Learning Term Semantics*

To capture the semantics of software-specific terms, we adopt the continuous skip-gram model [35], [36] which is the state-of-the-art algorithm for learning distributed word vector representations (or word embeddings) using a neural network model. The underling intuition of the algorithm is that words of similar meaning would appear in similar context. Therefore, the representation of each word can be defined on the words it frequently co-occurs with.

As illustrated in Figure 3, the objective of the continuous skip-gram model is to learn the word representation of each word that is good at predicting the surrounding words in the sentence. Formally, given a training sentence of K words $w_1, w_2, ..., w_K$, the objective of the continuous skip-gram model is to maximize the following average log probability:

$$L = \frac{1}{K} \sum_{k=1}^{K} \sum_{-N \preceq j \preceq N, j \neq 0} \log p(w_{k+j} | w_k) \quad (3)$$

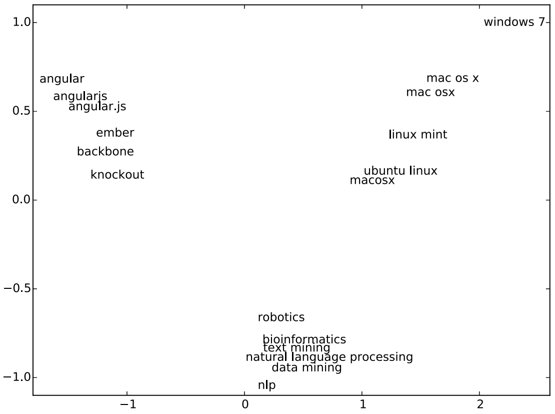


Fig. 4: An illustration of term representations in the word embedding space, two-dimensional projection using PCA.

where $w_k$ is the central word in a sliding window of the size $2N + 1$ over the sentence, $w_{k+j}$ is the context word surrounding $w_k$ within the sliding window. Our approach trains the continuous skip-gram model using the software-specific corpus obtained in Section III-B. We set the sliding window size $N$ at 5 in this work. That is, the sliding window contains 10 surrounding terms as the context terms for a given term in the sentence.

The probability $p(w_{k+j}|w_k)$ in Eq. 3 can be formulated as a log-linear softmax function which can be efficiently solved by the negative sampling method [36]. After the iterative feed-forward and back propagation, the training process finally converges, and each term obtains a low-dimensional real-valued vector (i.e., word embedding) in the resulting vector space. Following the experiments in our previous work to learn word embeddings from Stack Overflow corpus [24], we set the dimension of word embeddings at 200.

Stack Overflow is time-sensitive due to the evolution of technology landscape [14]. New terms emerge all the time, and existing term usage also changes over time. Word embedding is not good at encoding less frequent terms. If trained using the entire data, semantics of no-longer-actively-used or newly-appearing terms may not be well captured. To mitigate this issue, we split the Stack Overflow corpus into $M$ bulks of data evenly ($M = 11$ in this work, about 2.4 million posts per bulk). For each bulk of data $b_i$ ($1 \leq i \leq M$), we apply the continuous skip-gram model to the data and obtain a corresponding vector space $V_i$.

### E. Extracting Semantically Related Terms

For each software-specific term $t$ in the software-specific vocabulary, if the term $t$ is in the vector space $V_i$ ($1 \leq i \leq M$), we find a list of semantically related terms whose term vectors $v(w)$ are most similar to the vector $v(t)$ in the vector space using the following equation:

$$\operatorname*{argmax}_{w \in A_{V_i}} \cos(v(w), v(t)) = \operatorname*{argmax}_{w \in A_{V_i}} \frac{v(w) \cdot v(t)}{\|v(w)\| \|v(t)\|} \qquad (4)$$

where $A$ is the set of all terms in the vector space $V_i$ excluding the term $t$, and $\cos((v(w), v(t)))$ is the cosine similarity of the two vectors.

For a term $t \in V_i$, we select the top-20 most similar terms in the vector space $V_i$ as the candidate semantically related terms. As we split the whole corpus into $M$ bulks, we obtain $M$ vector spaces. Let $X$ be a set of vector spaces that contains the term $t$ ($1 \leq |X| \leq M$). Therefore, we obtain $|X|$ candidate lists for the term $t$. These $|X|$ candidate lists could overlap. We merge the $|X|$ candidate lists into one list and re-rank the candidate terms $w$ based on the equation:

$$semsim(w, t) = \frac{\sum_{V_i \in Y} \cos_{w \in V_i}(v(w), v(t))}{|Y|} \times \log_{|X|} |Y| \qquad (5)$$

where $Y$ is a set of vector spaces that contain both the candidate term $w$ and the term $t$ ($1 \leq |Y| \leq |X|$). The semantic similarity of the candidate term $w$ to the term $t$ is proportional to the two components: first, the average of the $w$'s cosine similarity with the term $t$ in the $|Y|$ vector spaces, and second, the logarithm of $|Y|$ on the base $|X|$. In practice, we add 1 to $|X|$ so that the log base is not 1 and add 1 to $|Y|$ so that the log will not be 0 when $|Y| = 1$. We take the logarithm of $|Y|$ on the base $|X|$ so that the two components in the equation have comparable contributions. This equation lessens the importance of some terms $w$ which may only be highly related to the term $t$ in a small number of vector spaces. Meanwhile, the less frequently used terms will not be overwhelmed by the more frequently used terms. We select the top-20 candidate terms in the reranked list as the semantically related terms for the term $t$.

Fig. 4 illustrates the set of semantically related terms for the three terms *angularjs*, *mac os x* and *natural language processing*. In the Figure, for the sake of clarity, we list only the top six most similar terms for the three terms respectively. These terms are projected into a two-dimensional vector space using Principal Component Analysis (PCA) [27], a technique commonly used to visualize high-dimensional vectors. We can see that semantically related terms are close to each other in the vector space. Furthermore, we can observe three kinds of relations between semantically related terms, 1) synonyms, e.g., (*angular, angular.js*), (*mac os x, macosx*); 2) abbreviations, e.g., (*natural language processing, nlp*); and 3) general relatedness, e.g., (*max osx, ubuntu linux*), (*angularjs, ember*), and (*nlp, data mining*). In this work, we focus on abbreviations and synonyms (referred to as morphological forms of a term in this work) among semantically related terms. Generally related terms could be also useful for recommendation systems, but they are out of the scope of this paper.

### F. Discriminating Synonyms & Abbreviations

We now explain the lexical rules and the string edit distance we use to discriminate synonyms and abbreviations of a term from its semantically related terms.

*1) Discriminating Morphological Synonyms:* In this work, we define synonyms as pairs of morphological similar terms.

TABLE II: Example Abbreviations and Synonyms

| RepTerm | Abbreviations | Synonyms |
|---|---|---|
| applicationcache | appcache | application cache |
| android-query | aquery | android query, androidquery |
| codeigniter | ci | codeingiter, codeignitor |
| algorithm | algo, algos | algoritms, algoritm |
| blackberry 10 | bb10, bb 10 | blackberry10 |

Some morphological-synonyms can be determined by stemming, such as (*object, objects*), (*rebase, rebasing*), but many other cannot, such as (*objective-c, objective c*), (*mac os x, macosx*), (*algorithm, algoritm (a misspelling)*), (*angular, angularjs*). We observe that morphological-synonyms among semantically related terms usually can be transformed from one term to another by a small number of string edits. Therefore, given a term $t$ and a semantically related term $w$, we use string edit distance to determine whether the two terms are morphological-synonyms.

Levenshtein distance [31] is often used to compute the string edit distance i.e., the minimum number of single-character edits (insertions, deletions or substitutions) required to transform one word into another. In this work, we use an enhanced string edit distance, Damerau-Levenshtein distance [18] (DL distance) to compute the minimum number of single-character edits (insert, delete, substitute, and transposition) required to transform one term to another. DL distance enhances the Levenshtein distance [31] with the transposition of the two adjacent characters such as ***fal**se* and ***fla**se*. Such character transpositions are a common source of misspellings. DL distance can more reliably detect such misspellings than the Levenshtein distance.

The absolute DL distance cannot be directly adopted for measurement. For example, the DL distance between *subdomain* and *sub-domains* and the DL distance between *jar* and *jsp* are both 2. The pair (*subdomain*, *sub-domains*) is morphological synonyms, while the pair (*jar*, *jsp*) is not. Therefore, we take into consideration both the absolute distance and the relative similarity between two term. For the absolute distance, the DL distance of the two synonyms must not be greater than 4, for example, the pair (*dispatcher.begininvoke, dispatcher.invoke*) will not be regarded as synonyms because the absolute DL distance between the two terms is 5.

For the relative similarity, we normalize the DL distance according to the maximum length of the two terms by:

$$similarity_{morph}(t, w) = 1 - \frac{DLdistance(t, w)}{max(len(t), len(w))} \tag{6}$$

The relative similarity indicates that the different parts of the two synonyms should be relatively small compared with the same parts of the two terms. In this work, we set the relative similarity threshold at $\frac{1}{3}$. As a result, the pair (*subdomain*, *sub-domains*) will be recognized as synonyms, but the pair (*jar*, *jsp*) will not, because the first pair is relatively similar enough, but the second pair is not.

*2) Discriminating Abbreviations:* If a semantically related term $w$ does not satisfy the requirement of being a synonym of a given term $t$, we further check whether it is an abbreviation of the given term. We consider the semantically related term $w$ as an abbreviation of the term $t$ if the they satisfy the following heuristics-based lexical rules. Similar rules are used to expand to identify abbreviations [30], [28].

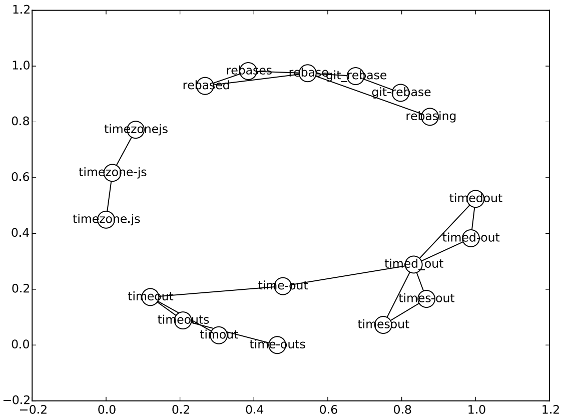


Fig. 5: The synonym graph: the edge represents the synonym relationship and a connected component represents a group of synonyms (zoom-in to view the details)

- The characters of the term $w$ must be in the same order as they appear in the term $t$, such as (*pypi*, ***py**thon **p**acage **i**ndex*), (*amq*, ***a**ctive**mq***);
- The length of the term $w$ must be shorter than that of the term $t$;
- If there are digits in the term $w$, there must be the same digits in the term $t$. For example, *vs2010* is regarded as an abbreviation of *visual studio 2010*, but *vs* is not regarded as an abbreviation of *visual studio 2010*;
- The term $w$ should not be the abbreviation of only some words in a multi-words phrase. For example, *cmd* is regarded as the abbreviation of *command*, but not as the abbreviation of *command line*.

It is important to note that we discriminate morphological synonyms and abbreviations from highly semantically related terms established by the terms' word embeddings. The above edit distance and lexical rules alone cannot reliably detect morphological synonyms and abbreviations without considering semantic relatedness between terms. For example, according to the above lexical rules, *ie* can be regarded as an abbreviation of *v**ie**w*. However, once considering semantic similarity, the term *ie* is not semantically related to the term *view*. Thus, *ie* will not even be an abbreviation candidate for *view*. Similarly, by solely DL distance, the terms (*opencv, opencsv*) will be regarded as synonyms. However, in our approach the two terms are not semantically related, and thus neither of them will be considered as synonym candidate for the other term.

### G. Grouping Morphological Synonyms

We identify synonyms for each term in our software-specific vocabulary. It is likely that we obtain separate but overlapping sets of synonyms for different terms. For example. for the

TABLE III: The coverage of terms in SOtag and CPtag datasets

| Dataset | #Term | #CoveredTerm | Coverage |
|---|---|---|---|
| SOtag | 21950 | 15385 | 70.1% |
| CPtag | 2391 | 1893 | 79.2% |

term *timed-out*, we obtain {*timedout, timed out*}, while for the term *timeout*, we obtain {*timout, timeouts, timed out*}. Note that the term *timed out* is in the two synonym sets. We group such overlapping sets of morphological synonyms for different terms into one set of morphological synonyms in which each pair of terms can be regarded as morphological synonyms.

To group separate but overlapping sets of morphological synonyms, we first build a graph of morphological synonyms based on the synonym relations between terms. Then, we find connected components in the graph as groups of morphological synonyms. Each pair of terms in a group is considered as synonyms. Figure 5 shows some examples[1]. For example, the term *timesout* is regarded as a synonym of *timedout* via the term *timed out*.

Considering all terms in a connected component as mutual synonyms, we essentially consider each group of morphological synonyms as a distinct concept. We select the term in the group with the highest usage frequency in Stack Overflow as the representative term of the concept. For each group of morphological synonyms (i.e., each concept), we merge the list of abbreviations of the terms in the group into a list of abbreviations for the group. Table II presents some examples of the representative terms and their abbreviations and synonyms identified by our approach.

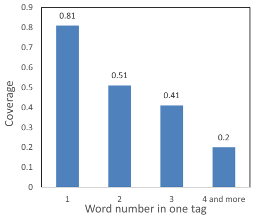
## IV. Evaluation of Our Thesaurus

In this section, we evaluate the coverage of software-specific vocabulary and the coverage of abbreviations and synonyms in our thesaurus $SEthesaurus$ against several community-created lists of computing-related terms, abbreviations and synonyms. We also manually examine the correctness of the identified abbreviations and synonyms. The evaluation confirms the effectiveness of our approach, meanwhile reveals potential enhancements of our current approach.

### A. The Coverage of Software-Specific Vocabulary

Our thesaurus contains 52,645 software-specific terms. To confirm whether our thesaurus covers a good range of software specific terms, we compare the software-specific vocabulary of our thesaurus against the three community-curated sets of software-specific terms.

In Stack Overflow, each question is tagged with up to five terms that describe the programming techniques and concepts of the question. These tags can be regarded as software-specific terms. Considering the power law distribution of tag usage, we consider tags that are used at least 30 times to avoid rare terms, and we collect in total 21,950 tags (as of August 2015) from over 9-millions questions to check if these tags are in our vocabulary. In addition, we also collect 2,391 tags from 263-thousands questions in the other programming Q&A site, CodeProject. For brevity, we refer these two datasets as *SOtag* and *CPtag*.



| Frequency | #Tags | Coverage |
|---|---|---|
| 30∼100 | 9497 | 54.6% |
| 100∼1000 | 7555 | 78.8% |
| 1000∼10000 | 2463 | 91.4% |
| >10000 | 406 | 96.6% |

Fig. 6: The barchart shows the coverage for tags with different number of words. The table shows that coverage varies with tag frequency.

Our software-specific vocabulary covers 70.1% terms in the SOtag dataset, and 79.2% terms in the CPtag dataset. By observing the Stack Overflow and CodeProject tags, we further find three reasons why some tags are not covered by our software-specific vocabulary. We explain our observations using Stack Overflow dataset.

First, some tags contain four or more words, and many of them contain version number such as *google-maps-api-3* and *ruby-on-rails-3.2*. However, people often do not mention version numbers when mentioning a technique in discussions. Therefore, our vocabulary may not contain a specific version of a technique, but it usually contain the general term for the technique, such as *google maps api*, *ruby on rails*. As shown in the bar chart of Fig 6, the coverage for the tags with 4 or more words is low (about 20%). However, for the tags with 3 or less words, the coverage becomes much higher. Second, as shown in the table of Fig 6, for tags that are used more than 1000 times, the coverage by our vocabulary can reach 90% or higher. But for tags that are used less than 100 times, the coverage is only about 54.6%. Note that although less frequently-used tags (30-1000 times) account for 86% of the tags, their total times of usage in Stack Overflow account for only 6.1% of the total tag usage. Therefore, the impact of missing some less frequently used tags (especially those used 30-100 times) on NLP tasks like information retrieval is minor. Third, some tags are artificial terms for question tagging, such as *android-asynctask* and *django-views*, but these terms are rarely used in discussion text.

### B. Abbreviation Coverage

Our approach finds 4,773 abbreviations for 4,234 terms (one term may have several abbreviations) from Stack Overflow corpus. In Wikipedia, there is a list of computing and IT abbreviations [1]. The list contains 1,292 full names and each full name has one abbreviation, except for *regular expression* which has two abbreviations. 855 of these 1,292 full names are in our vocabulary. For those 437 full names that are not in our vocabulary, they are either long phrases (e.g., *context and dependency injection*, *advanced data communications control procedures*) or related to other domains such as communication (e.g., *atm adaptation layer, advanced research projects agency*), and thus are not mentioned frequently enough in

[1] For multi-words phrases (e.g., *git rebase*), we replace space with "_" for the visualization clarity.

Stack Overflow for our approach to identify them as software-specific terms.

We use the 855 full names and their abbreviations as ground truth to examine the coverage of the identified abbreviations in our thesaurus. 751 of these 855 full names have abbreviations in our thesaurus, and the abbreviations of 739 out of the 751 terms are in the ground truth. That is, the accuracy is 86%. According to our observation, two reasons result in the missing abbreviations. First, there are some unusual abbreviations in the Wikipedia list which we believe developers more like to use full names instead of the abbreviations, e.g., *access time* instead of *at*. Second, there are limitations with our abbreviation inference heuristics which cannot find abbreviations with unique string transformations, such as *i18n* for *internationalization*, *xss* for *cross-site scripting*, and *w3c* for *world wide web consortium*. In fact, our approach identifies these abbreviations as semantically related to their full names. However, due to their unique string transformation, general lexical rules cannot determine them as abbreviations.

Compared with the Wikipedia abbreviation list, our dictionary contains much more software-specific terms and more abbreviations, for example, *abc* for *abstract base class*, *sso* for *single sign-on*. Furthermore, our approach can capture multiple abbreviations for a term. For example, our approach finds 7 abbreviations (*regex, reg exp, regexps, regexes, regexp, regexs, reg-ex*) for *regular expression* in the Stack Overflow text, while the Wikipedia list include only two of these 7 abbreviations.

### C. Synonym Coverage

Our approach identifies 14,006 synonym groups which contain 38,104 morphological terms[2]. To examine the coverage and accuracy of the identified synonyms, we compare our results against the Stack Overflow tag synonyms. Stack Overflow users collaboratively maintain a list of tag synonym pairs. By August 2015, there are 3,231 community-approved synonym pairs [5]. Each pair has a synonym tag and a master tag. We take these tag synonym pairs as the ground truth. According to Stack Overflow tag naming convention, multi-words in a tag are concatenated by "-", while in plain text, users more likely write them with spaces. Thus, we replace "-" in the tag with space for this comparison, for example, the tag *visual-studio* will be transformed into *visual studio*.

For each synonym tag (e.g., *videojs*) in the ground truth, we check if it is in a synonym group that our approach identifies, and if so, we further check if its corresponding master tag (e.g., *video.js*) is also in the synonym group. As Stack Overflow tag synonyms sometimes involve abbreviations, such as (*js, javascript*), we also check if a synonym tag is an abbreviation of a synonym group and if the master tag is in the corresponding synonym group.

We compare our approach with the two baselines, Wordnet and SEWordSim. WordNet [37] is a general-purpose lexical database of English created by lexicographers. WordNet groups English words into synonym sets (synsets) such as {*small, little, minor*}. For each synonym tag in the ground truth, we check if it is in the WordNet, and if so, we further check if the master tag is in the same synset as the synonym tag in the WordNet. SEWordSim [52] is a software-specific word similarity database that is extracted from Stack Overflow. For each synonym tag in ground truth, we check if it is in the SEWordSim database, and if so, we further check if the master tag is in the list of the top-20 most similar words for the synonym tag in the SEWordSim database.

TABLE IV: The coverage of synonyms in three methods.

| Method | #CoveredSynonym | #CoveredMaster | Accuracy |
|---|---|---|---|
| SEthesaurus | 2,316 | 1,439 | **62.1**% |
| WordNet | 725 | 218 | 30.7% |
| SEWordSim | 941 | 86 | 9.1% |

Table IV summarizes the results. Overall, 2,316 (71.7%) out of 3,231 synonym tags are covered by our synonym groups, while only 725 (22.4%) and 941 (29.1%) are contained in the WordNet and the SEWordSim database. Out of the 2,316 synonym tags, 1,439 (62.1%) correct synonyms are contained in our synonym groups. This significantly outperform the accuracy of the WordNet synonyms (30.7%) and the SEWordSim synonyms (9.1%).

We further explore why our approach misses 877 (2,316-1,439) tag synonyms. First, some synonym pairs are not morphological which is beyond our scope, such as (*sky, flutter*) and (*wallet, passbook*). Second, the Stack Overflow community sometimes merge fine-grained concepts into more general ones as tag synonyms, such as (*css-reset, css*), (*flash-player, flash*) and (*worksheet, excel*). However, such fine-grained terms and general terms have different meanings in the discussion text, and our approach do not regard them as synonyms.

### D. Human Evaluation

As shown in the above evaluation, compared with several community-curated ground truth, our thesaurus contains much more software-specific terms, and a term in our thesaurus often has several abbreviations and synonyms. Therefore, our evaluation against these community-curated ground truth shows only the correctness of a subset of abbreviations and synonyms that our approach identifies, but it does not show whether many other abbreviations and synonyms that are not included in the ground truth are correct or not.

To verify the general correctness of the abbreviations and synonyms in our thesaurus, we recruit four participants for the manual evaluation including 3 final-year undergraduate students and one research assistant with master degree majoring in computer science. They all have several-year programming experience. We split them into two groups. For each group, we randomly sample 200 abbreviation pairs and 400 synonym pairs in our thesaurus for the evaluation. Each participant independently examines the assigned samples without any discussions. They judge the correctness of abbreviations and synonyms based on their knowledge, as well as the Wikipedia and other available online information. To avoid bias, we count only pairs which are marked as correct by both participants

[2] Some terms do not have abbreviations or synonyms

in a group as the correct ones. In total, 400 abbreviations and 800 synonyms are manually examined.

The human evaluation confirms that 297 (74.3%) abbreviation pairs and 686 (85.8%) synonym pairs are correct. We further investigate the reasons for those incorrect pairs. Two reasons result in the wrong abbreviation pairs. First, the rules described in Section III-F could erroneously classify terms as abbreviations, such as *istream* as the abbreviation of *inputstream*, or *64-bit os* as the abbreviation of *64-bit windows*. These pairs of terms are semantically similar, but they are not abbreviations. Second, some abbreviation errors are caused by erroneous synonyms and synonym grouping. For example, *btle* is the abbreviation of *bluetooth le* (bluetooth low energy). Our approach erroneously recognizes *bluetooth le* as the synonym of *bluetooth*. Consequently, *btle* is erroneously regarded as an abbreviation of *bluetooth*. For synonyms, most errors are caused by term pairs that are both semantically and lexically similar, but are not synonyms, such as (*minsdkversion, maxsdkversion*), (*notification bar, notification tray*) and (*schema.xml, schema.yml*). Other synonym errors are also caused by erroneous synonyms and synonym grouping, similar to the example of the abbreviation error (*btle, bluetooth*).

## V. Usefulness Evaluation

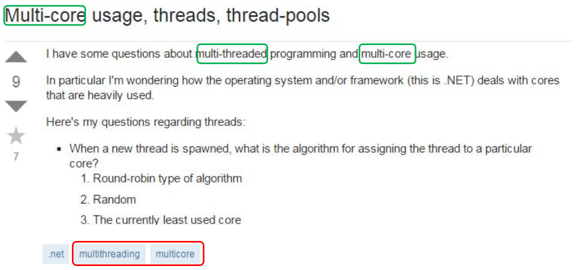
After evaluating the quality of our thesaurus, we now demonstrate the usefulness of our thesaurus for text normalization tasks.

### A. Background

NLP-based techniques have been widely used to support software engineering tasks involving text data [22], [40], [32]. As abbreviations and synonyms are commonly used in software engineering text, normalizing these abbreviations and synonyms becomes one of the fundamental steps to achieve high quality text mining results [30], [29]. Abbreviations and synonyms are often referred to as inflected (or derived) words in natural language processing. The goal of text normalization is to reduce inflected (or derived) words to their root form. Techniques developed for general English text, such as stemming [45] or WordNet lemmatization [7], are commonly adopted for software engineering text. Some work proposes domain-specific techniques to normalize source code vocabulary (e.g., expanding abbreviated identifiers), but none of existing work examines the normalization of informal software engineering text on social platforms.

### B. Experiment Setup

*1) Dataset:* We randomly sample 100,000 questions from Stack Overflow. To further demonstrate the generality of our thesaurus, we also randomly sample 50,000 questions from CodeProject[3], which is another popular Q&A web site for computer programming. We preprocess the sampled questions in the same way as described in Section III-B.



Multi-core usage, threads, thread-pools

I have some questions about multi-threaded programming and multi-core usage.

In particular I'm wondering how the operating system and/or framework (this is .NET) deals with cores that are heavily used.

Here's my questions regarding threads:

- When a new thread is spawned, what is the algorithm for assigning the thread to a particular core?
  1. Round-robin type of algorithm
  2. Random
  3. The currently least used core

Fig. 7: The tags and their different forms in the question

*2) Compared Methods:* The task is to normalize the title and content of the sampled questions. We develop a software-specific lemmatizer powered by our thesaurus for normalizing abbreviations and synonyms in informal software engineering text. We compare the performance of our lemmatizer with the two baseline methods that are commonly used for text normalization, i.e., Porter stemming [45] and WordNet-based lemmatization [7]. For our lemmatizer, we reduce abbreviations and synonyms to their representative terms in our thesaurus. Porter stemming reduces inflected (or derived) words to their stems by removing derivational affixes at the end of the words. WordNet-based lemmatization reduces different forms of a word to their lemma based on WordNet synset (i.e., set of synonyms created by highly trained linguists).

*3) Ground Truth and Evaluation Metrics:* We adopt question tags as the ground truth to evaluate the effectiveness of the text normalization. Question tags can be considered as metadata of question text. We normalize question tags in the same ways as we normalize question title and content using the three compared methods. Then, we measure the effectiveness of a text normalization method by how much percentage of tags appear in question title and content before and after text normalization. We take an average of the percentage over all the sampled questions. Essentially, we investigate how much text normalization can make question texts more consistent with question metadata. Fig. 7 shows an example. Before text normalization, only one of the three tags (*.net*) appears in the question title and content. After normalization using our lemmatizer, all 3 tags appear in the question title and content.

### C. Results

As shown in Fig. 8, without text normalization, on average only 55.5% and 54.0% tags appear in the title and content of the sampled Stack Overflow and CodeProject questions, respectively. This indicates that the consistency between question texts and question metadata is low. With text normalization by our lemmatizer, the percentage is boosted to 79.3% for the sampled Stack Overflow questions, and 68.7% for the sampled CodeProject questions. Although Porter stemming and WordNet-based lemmatization can also improve the consistency between question texts and question metadata, the improvement in percentage is much smaller or only marginally, compared with our lemmatizer.

The Porter stemming can only find words with derivational affixes such as (*upload, uploaded*) or singular and plural forms

[3] http://www.codeproject.com/script/Answers/List.aspx??tab=active

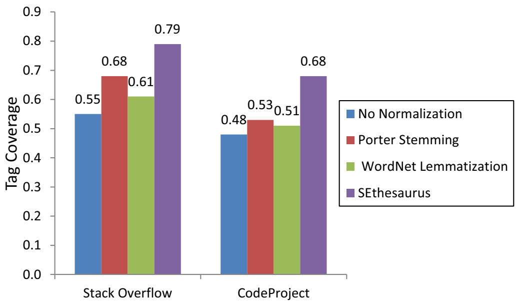


Fig. 8: The average percentage of tags appearing in question text by different normalization methods

such as (*script, scripts*). The WordNet-based lemmitization can recognize more synonyms based on WordNet synset, such as (*os, operating system*). However, WordNet is a general thesaurus and lacks many software-specific terms. In contrast, our thesaurus is mined from the vast amount of software engineering text and contain a much richer set of software-specific terms and their abbreviations and synonyms. Furthermore, our thesaurus can recognize complicated synonyms, such as (*multithreading, multi-thread*) and (*windows, windwos*) that are difficult to find using Porter stemming and WordNet lemmatization. Therefore, our domain-specific thesaurus is more suitable for software-specific text normalization than general stemming methods or general English thesaurus.

## VI. Tool Support

We build a simple demonstration website[4] for the community to access to our thesaurus. In the website, users can search for software-specific terms and find their abbreviations and synonyms. As our thesaurus mainly aims at automatic text mining tasks, we also release the API (similar to WordNet API) to access to the information in the thesaurus.

## VII. Discussion

Finally, we discuss the potential enhancement of our approach and some extension applications.

### A. Enhancements of Our Approach

Our evaluation reveals some limitations of our approach which could be enhanced in the future. First, our approach may miss some long but infrequent phrases (e.g., *context and dependency injection*), common-word software-specific terms (e.g., *seahorse*), and software-specific terms that are neither frequently mentioned in the software-specific corpus nor in the general corpus. To address these issues, we could incorporate semi-supervised methods for software-specific named entity recognition proposed in our recent work [57], [58].

Second, our approach currently learns and analyzes term semantics. However, we find that the same term can have different senses in different contexts. For example, the term *post* is both an abbreviation of *power on self test* and a normal term that refers to online post. Furthermore, a term can be the abbreviation of several terms, for example, *bcp* for *bulk copy* and *best current practice*, *apt* for *advanced packaging tool* and *annotation processing tool*. To disambiguate context-sensitive term semantics, we need to learn term semantics at the sense level [25] i.e., considering the context around the term.

TABLE V: Misspelling examples in our thesaurus

| Term | Misspellings |
|---|---|
| ubuntu | ubunutu |
| jquery | jqeury, jquey |
| eclipse | eclispe, eclise, eclips, eclipe |
| android | anroid, andoid, andriod, adroid, andorid |
| bootstrap | bootstarp, bootstap, boostrap, bootsrap |
| postgresql | postgressql, postresql, posgresql, postgesql |

### B. Domain-Specific Spelling Checking

While developers write natural language documents, such as comments, documentations, blogs and Q&A posts, it is very natural that misspellings occur. In fact, a large portion of morphological synonyms that our approach identifies in Stack Overflow text are misspellings (see Table V for examples in our thesaurus). To avoid misspellings, developers can run spell checking in their editors or IDEs. Some researchers [41], [21] also use spell checkers (e.g., Aspell [2], Hunspell [3]) to pre-processing software engineering text. However, existing spell checkers are trained for general English text, without the knowledge about software-specific terms and their common misspellings such as the examples in Table V.

Unlike Beyer and Pinzger's work [8] in which they observe synonym transformation rules manually from a small dataset, we could extend our approach with neural network techniques [17], [10], [34] to train a software-specific spelling checking tool based on the large-scale dataset. This domain-specific spell checker can help developers correct misspellings while writing a document or help researchers normalize software engineering text for high quality text mining [41].

## VIII. Conclusions and Future Work

In this paper, we present an automatic approach for mining a thesaurus of software-specific terms and commonly-used morphological forms from informal software engineering discussions. Our evaluation shows that our thesaurus covers a large set of software-specific terms, abbreviations and synonyms with high accuracy. In a text normalization task, we demonstrate that our thesaurus can significantly improve the consistency between question text and question metadata, compared with general stemming and lemmatization methods. In the future, we will extend our thesaurus into a software-specific dictionary for software engineering text (simile to WordNet) which can enable many applications in software engineering, such as domain-specific spelling checking, recommendation of semantically related techniques, software-text cleaning, etc.

## Acknowledgment

The authors would like to thank Prof Yang Liu for his discussions and support. Due to the author limit policy, we cannot include him as a co-author of this work. This work was partially supported by MOE AcRF Tier1 Grant M4011267.020 and M4011448.020.

[4] https://se-thesaurus.appspot.com/

## REFERENCES

[1] Abbreviations in wikipedia. https://en.wikipedia.org/wiki/List_of_computing_and_IT_abbreviations. Accessed: 2016-06-20.

[2] Gnu aspell. http://aspell.net/.

[3] Hunspell. https://hunspell.github.io/.

[4] Stack overflow data dump. https://archive.org/details/stackexchange. Accessed: 2016-02-20.

[5] Tag synonyms in stack overflow. http://stackoverflow.com/tags/synonyms. Accessed: 2016-06-20.

[6] Wikipedia data dump. https://dumps.wikimedia.org/enwiki/latest/. Accessed: 2016-02-20.

[7] Wordnet lemmatization. http://www.nltk.org/_modules/nltk/stem/wordnet.html.

[8] S. Beyer and M. Pinzger. Synonym suggestion for tags on stack overflow. In *Proceedings of the 2015 IEEE 23rd International Conference on Program Comprehension*, pages 94–103. IEEE Press, 2015.

[9] S. Beyer and M. Pinzger. Grouping android tag synonyms on stack overflow. In *Proceedings of the 13th International Workshop on Mining Software Repositories*, pages 430–440. ACM, 2016.

[10] J. A. Botha and P. Blunsom. Compositional morphology for word representations and language modelling. In *ICML*, pages 1899–1907, 2014.

[11] C. Chen, S. Gao, and Z. Xing. Mining analogical libraries in q&a discussions–incorporating relational and categorical knowledge into word embedding. In *2016 IEEE 23rd International Conference on Software Analysis, Evolution, and Reengineering (SANER)*, volume 1, pages 338–348. IEEE, 2016.

[12] C. Chen and Z. Xing. Mining technology landscape from stack overflow. In *Proceedings of the 10th ACM/IEEE International Symposium on Empirical Software Engineering and Measurement*, page 14. ACM, 2016.

[13] C. Chen and Z. Xing. Similartech: automatically recommend analogical libraries across different programming languages. In *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering*, pages 834–839. ACM, 2016.

[14] C. Chen and Z. Xing. Towards correlating search on google and asking on stack overflow. In *40th IEEE Computer Society International Conference on Computers, Software & Applications (COMPSAC)*, volume 1, pages 83–92. IEEE, 2016.

[15] C. Chen, Z. Xing, and L. Han. Techland: Assisting technology landscape inquiries with insights from stack overflow. In *Proceedings of the 32nd IEEE International Conference on Software Maintenance and Evolution*, 2016.

[16] A. Corazza, S. Di Martino, and V. Maggio. Linsen: An efficient approach to split identifiers and expand abbreviations. In *Software Maintenance (ICSM), 2012 28th IEEE International Conference on*, pages 233–242. IEEE, 2012.

[17] M. Creutz and K. Lagus. Unsupervised morpheme segmentation and morphology induction from text corpora using morfessor 1.1. *Publications Comput. Inf. Sci., Helsinki Univ. Technol., Tech. Rep. A*, 81:2005, 2005.

[18] F. J. Damerau. A technique for computer detection and correction of spelling errors. *Communications of the ACM*, 7(3):171–176, 1964.

[19] B. Dit, L. Guerrouj, D. Poshyvanyk, and G. Antoniol. Can better identifier splitting techniques help feature location? In *Program Comprehension (ICPC), 2011 IEEE 19th International Conference on*, pages 11–20. IEEE, 2011.

[20] E. Enslen, E. Hill, L. Pollock, and K. Vijay-Shanker. Mining source code to automatically split identifiers for software analysis. In *2009 6th IEEE International Working Conference on Mining Software Repositories*, pages 71–80. IEEE, 2009.

[21] A. C. Finkelstein, D. Gabbay, A. Hunter, J. Kramer, and B. Nuseibeh. Inconsistency handling in multiperspective specifications. *Software Engineering, IEEE Transactions on*, 20(8):569–578, 1994.

[22] M. Gegick, P. Rotella, and T. Xie. Identifying security bug reports via text mining: An industrial case study. In *Mining software repositories (MSR), 2010 7th IEEE working conference on*, pages 11–20. IEEE, 2010.

[23] L. Guerrouj, P. Galinier, Y.-G. Guéhéneuc, G. Antoniol, and M. Di Penta. Tris: A fast and accurate identifiers splitting and expansion algorithm. In *2012 19th Working Conference on Reverse Engineering*, pages 103–112. IEEE, 2012.

[24] C. Guibin, C. Chen, Z. Xing, and X. Bowen. Learning a dual-language vector space for domain-specific cross-lingual question retrieval. In *31st IEEE/ACM International Conference on Automated Software Engineering (ASE). IEEE/ACM*, 2016.

[25] Y. HaCohen-Kerner, A. Kass, and A. Peretz. Combined one sense disambiguation of abbreviations. In *Proceedings of the 46th Annual Meeting of the Association for Computational Linguistics on Human Language Technologies: Short Papers*, HLT-Short '08, pages 61–64, Stroudsburg, PA, USA, 2008. Association for Computational Linguistics.

[26] M. J. Howard, S. Gupta, L. Pollock, and K. Vijay-Shanker. Automatically mining software-based, semantically-similar words from comment-code mappings. In *Proceedings of the 10th Working Conference on Mining Software Repositories*, pages 377–386. IEEE Press, 2013.

[27] I. Jolliffe. *Principal component analysis*. Wiley Online Library, 2002.

[28] D. Lawrie and D. Binkley. Expanding identifiers to normalize source code vocabulary. In *Software Maintenance (ICSM), 2011 27th IEEE International Conference on*, pages 113–122. IEEE, 2011.

[29] D. Lawrie, D. Binkley, and C. Morrell. Normalizing source code vocabulary. In *2010 17th Working Conference on Reverse Engineering*, pages 3–12. IEEE, 2010.

[30] D. Lawrie, H. Feild, and D. Binkley. Extracting meaning from abbreviated identifiers. In *Seventh IEEE International Working Conference on Source Code Analysis and Manipulation (SCAM 2007)*, pages 213–222. IEEE, 2007.

[31] V. I. Levenshtein. Binary codes capable of correcting deletions, insertions, and reversals. In *Soviet physics doklady*, volume 10, pages 707–710, 1966.

[32] E. Linstead, P. Rigor, S. Bajracharya, C. Lopes, and P. 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.

[33] H. Liu, Z.-Z. Hu, J. Zhang, and C. Wu. Biothesaurus: a web-based thesaurus of protein and gene names. *Bioinformatics*, 22(1):103–105, 2006.

[34] T. Luong, R. Socher, and C. D. Manning. Better word representations with recursive neural networks for morphology. In *CoNLL*, pages 104–113, 2013.

[35] T. Mikolov, K. Chen, G. Corrado, and J. Dean. Efficient estimation of word representations in vector space. *arXiv preprint arXiv:1301.3781*, 2013.

[36] T. Mikolov, I. Sutskever, K. Chen, G. S. Corrado, and J. Dean. Distributed representations of words and phrases and their compositionality. In *Advances in neural information processing systems*, pages 3111–3119, 2013.

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

[38] E. Murphy-Hill, A. P. Black, D. Dig, and C. Parnin. Gathering refactoring data: a comparison of four methods. In *Proceedings of the 2nd Workshop on Refactoring Tools*, page 7. ACM, 2008.

[39] Y. Park, S. Patwardhan, K. Visweswariah, and S. C. Gates. An empirical analysis of word error rate and keyword error rate. In *INTERSPEECH*, pages 2070–2073, 2008.

[40] G. Petrosyan, M. P. Robillard, and R. De Mori. Discovering information explaining api types using text classification. In *Proceedings of the 37th International Conference on Software Engineering-Volume 1*, pages 869–879. IEEE Press, 2015.

[41] 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*, pages 94–125. Springer, 2013.

[42] L. Ponzanelli, A. Bacchelli, and M. Lanza. Seahawk: Stack overflow in the ide. In *Proceedings of the 2013 International Conference on Software Engineering*, pages 1295–1298. IEEE Press, 2013.

[43] S. P. Ponzetto and M. Strube. Deriving a large scale taxonomy from wikipedia. In *AAAI*, volume 7, pages 1440–1445, 2007.

[44] S. P. Ponzetto and M. Strube. Knowledge derived from wikipedia for computing semantic relatedness. *J. Artif. Intell. Res.(JAIR)*, 30:181–212, 2007.

[45] M. F. Porter. An algorithm for suffix stripping. *Program*, 14(3):130–137, 1980.

[46] D. Shepherd, Z. P. Fry, E. Hill, 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*, pages 212–224. ACM, 2007.

[47] D. Shepherd, L. Pollock, and K. Vijay-Shanker. Towards supporting on-demand virtual remodularization using program graphs. In *Proceedings of the 5th international conference on Aspect-oriented software development*, pages 3–14. ACM, 2006.

[48] R. Soricut and F. Och. Unsupervised morphology induction using word embeddings. In *Proc. NAACL*, 2015.

[49] G. Sridhara, E. Hill, L. Pollock, and K. Vijay-Shanker. Identifying word relations in software: A comparative study of semantic similarity tools. In *Program Comprehension, 2008. ICPC 2008. The 16th IEEE International Conference on*, pages 123–132. IEEE, 2008.

[50] S. Subramanian, L. Inozemtseva, and R. Holmes. Live api documentation. In *Proceedings of the 36th International Conference on Software Engineering*, pages 643–652. ACM, 2014.

[51] Y. Tian, D. Lo, and J. Lawall. Automated construction of a software-specific word similarity database. In *Software Maintenance, Reengineering and Reverse Engineering (CSMR-WCRE), 2014 Software Evolution Week-IEEE Conference on*, pages 44–53. IEEE, 2014.

[52] Y. Tian, D. Lo, and J. Lawall. Sewordsim: Software-specific word similarity database. In *Companion Proceedings of the 36th International Conference on Software Engineering*, pages 568–571. ACM, 2014.

[53] C. Treude and M. P. Robillard. Augmenting api documentation with insights from stack overflow. In *Proceedings of the 38th International Conference on Software Engineering*, pages 392–403. ACM, 2016.

[54] S. Wang, D. Lo, and L. Jiang. Inferring semantically related software terms and their taxonomy by leveraging collaborative tagging. In *Software Maintenance (ICSM), 2012 28th IEEE International Conference on*, pages 604–607. IEEE, 2012.

[55] B. Xu, D. Ye, Z. Xing, X. Xia, C. Guibin, and L. Shanping. Predicting semantically linkable knowledge in developer online forums via convolutional neural network. In *31st IEEE/ACM International Conference on Automated Software Engineering (ASE). IEEE/ACM*, 2016.

[56] J. Yang and L. Tan. Swordnet: Inferring semantically related words from software context. *Empirical Software Engineering*, 19(6):1856–1886, 2014.

[57] D. Ye, Z. Xing, C. Y. Foo, Z. Q. Ang, J. Li, and N. Kapre. Software-specific named entity recognition in software engineering social content. In *2016 IEEE 23rd International Conference on Software Analysis, Evolution, and Reengineering (SANER)*, volume 1, pages 90–101. IEEE, 2016.

[58] D. Ye, Z. Xing, C. Y. Foo, J. Li, and N. Kapre. Learning to extract api mentions from informal natural language discussions. In *2016 IEEE 23rd International Conference on Software Analysis, Evolution, and Reengineering (SANER)*, volume 1, pages 90–101. IEEE, 2016.

[59] X. Ye, H. Shen, X. Ma, R. Bunescu, and C. Liu. From word embeddings to document similarities for improved information retrieval in software engineering. In *Proceedings of the 38th International Conference on Software Engineering*, pages 404–415. ACM, 2016.

[60] Y. Zou, T. Ye, Y. Lu, J. Mylopoulos, and L. Zhang. Learning to rank for question-oriented software text retrieval (t). In *Automated Software Engineering (ASE), 2015 30th IEEE/ACM International Conference on*, pages 1–11. IEEE, 2015.