SEthesaurus: WordNet in Software Engineering

Xiang Chen, *Member, IEEE,* Chunyang Chen, *Member, IEEE,* Dun Zhang, and Zhenchang Xing, *Member, IEEE,*

Abstract—Informal discussions on social platforms (e.g., Stack Overflow, CodeProject) have accumulated a large body of programming knowledge in the form of natural language text. Natural language process (NLP) techniques can be utilized to harvest this knowledge base for software engineering tasks. However, consistent vocabulary for a concept is essential to make an effective use of these NLP techniques. Unfortunately, the same concepts are often intentionally or accidentally mentioned in many different morphological forms (such as abbreviations, synonyms and misspellings) in informal discussions. Existing techniques to deal with such morphological forms are either designed for general English or mainly resort to domain-specific lexical rules. A thesaurus, which contains software-specific terms and commonly-used morphological forms, is desirable to perform normalization for software engineering text. However, constructing this thesaurus in a manual way is a challenge task. In this paper, we propose an automatic unsupervised approach to build such a thesaurus. In particular, we first identify software-specific terms by utilizing a software-specific corpus (e.g., Stack Overflow) and a general corpus (e.g., Wikipedia). Then we infer morphological forms of software-specific terms by combining distributed word semantics, domain-specific lexical rules and transformations. Finally, we perform graph analysis on morphological relations. We evaluate the coverage and accuracy of our constructed thesaurus against community-cumulated 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 constructed thesaurus by developing three applications and also verify the generality of our approach in constructing thesauruses from data sources in other domains.

Index Terms—Software-specific Thesaurus, Natural Language Processing, Morphological Form, Word Embedding.

1 Introduction

Developers often discuss in the form of natural language text on social platforms (such as Stack Overflow, CodeProject) to share and acquire programming knowledge. Therefore, many natural-language-processing (NLP) based techniques have been proposed and developed to mine programming knowledge from such informal discussions. The mined knowledge can assist developers for many software engineering tasks, such as searching for documents [1] [2], categorizing software technologies [3], extracting API mentions and usage insights [4] [5], recovering traceability among informal discussions (e.g., duplicate questions [6]) or between code and informal discussions [7], linking domain-specific entities in informal discussions to official documents [8] and mining technology landscapes [9], [10]. To make an effective use of NLP techniques in these tasks, a consistently-used vocabulary of software-specific terms is essential, since NLP techniques assume that the same words are used whenever a particular concept is mentioned.

Since informal discussions are created by millions of developers, who have diverse technical and linguistic background, the same concept is often mentioned in many *morphological forms* (such as abbreviations, synonyms and misspellings) intentionally or accidentally [4]. Fig. 1 shows three posts from Stack Overflow, which discuss the slash issue of regular expression when parsing JavaScript. These three posts are marked as duplicate posts by the Stack Overflow community, since they discuss the same programming issue (i.e., the three posts are considered as semantically equivalent). However, when mentioning *regular expression* and *JavaScript*, these 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 1 summarizes the frequencies of various morphological forms of *convolutional neural network* in Stack Overflow discussions. Note that there exist many morphological forms for the same concept, and some forms are even used as frequently as the standard one (e.g., *convolutional neural network* and *cnn* in Table 1).

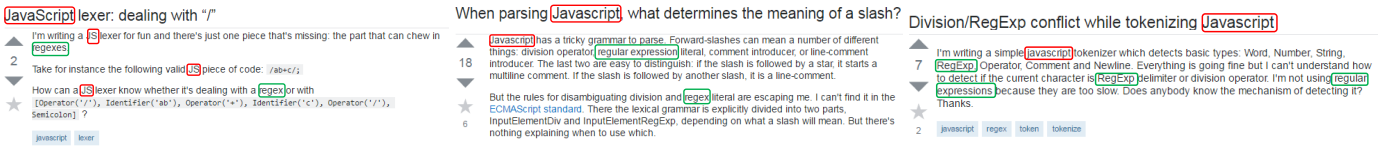
TABLE 1: Morphological forms of *convolutional neural network*

Term	Frequency	Annotation
convolutional neural network	992	Standard
cnn	5,901	Abbreviation
cnns	502	Abbreviation
convolutional-neural-networks	469	Synonym
convnet	376	Abbreviation
convolution-neural-network	185	Synonym
convnets	99	Abbreviation
convolution neural networks	33	synonym

The wide presence of morphological forms for the same concept in informal discussions results in a serious challenge for informal-retrieval based software engineering tasks. 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*

- *Xiang Chen is with the School of Information Science and Technology, Nantong University, China. E-mail: xchencs@ntu.edu.cn*
- *Chunyang Chen is with the Faculty of Information Technology, Monash University, Australia. E-mail: chunyang.chen@monash.edu*
- *Dun Zhang is is with the School of Information Science and Technology, Nantong University, China. E-mail: dunnzhang0@gmail.com*
- *Zhenchang Xing is with College of Engineering & Computer Science, Australian National University, Australia. E-mail: zhenchang.xing@anu.edu.au*

Manuscript received April 19, 2019; revised August 26, 2019.



(a) post ID=14553203 (b) post ID=5519596 (c) post ID=4726295

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

used in these posts, even though these three posts are semantically equivalent. Moreover, 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 have negative effects on the performance of the NLP techniques.

Word embedding and neural network [11], [12], [13] have been used in current NLP research domain to deal with the issue of morphology. A recent work by Soricut and Och [14] exploits the exhibited relational regularities via word embedding (e.g., *car* to *cars*, *dog* to *dogs*) to model prefix-based and suffix-based morphological rules and transformations. However, these morphology learning techniques in the NLP domain only consider morphological relations, which are drawn out of linguistic regularities of natural language (e.g., prefix and suffix). As shown in Fig. 1 and Table 1, 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*).

The impact of inconsistent vocabulary has been recognized for the application of NLP-based techniques to source code and software documentation [15], [16], [17], [18], [19] in the software engineering research domain. The focus has been put 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 and it is completely irrelevant to *opencv*, which is a computer vision library. To solve this issue, most of previous 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 of previous approaches [20], [21], [22] exploit word frequency in word co-occurrence data to rank abbreviation expansions, but none of them exploit semantic relevance 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 compares software engineering text (e.g., discussions from Stack Overflow) against general text (e.g., Wikipedia documents) to derive a vocabulary of software-specific terms, which are used in software engineering text. It then combines the state-of-the-art word embedding methods 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, which 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 finds groups of morphological forms, each expressing a distinct concept (examples can be found in Fig. 5), similar to the notion of synsets[1] provided by WordNet [23].

After compared with several community-cumulated lists of IT terms, abbreviations and synonyms, our approach can automatically infer software-specific terms and morphological forms, which are up-to-date and actively used in Stack Overflow. In contrast, community-cumulated lists contain many out-of-date terms and rarely-used morphological forms. This result demonstrates the necessity of our proposed approach and shows the advantage of our approach (i.e., it can automatically infer software-specific morphological forms). Examination in the manual way on randomly sampled 384 abbreviations and synonyms shows the high coverage (72.9% and 86.5%) of our approach. To demonstrate the usefulness of the inferred morphological forms of software-specific terms for information retrieval, we develop three applications including software-specific text normalization, identifying tag synonyms on Stack Overflow and software-specific spelling correction. In addition, the success of constructing thesaurus in other domain-specific datasets also shows the generality of our approach.

This paper is an extended version of our earlier study [24]. The extension makes the following additional contributions:

- We update the software-specific thesaurus based on the latest Stack Overflow data dump.
- Apart from the continuous skip-gram model [25], we also adopt FastText model [26] which takes subword information into the consideration to learn the fine-grained semantic relevance between software-specific terms.
- Based on the extracted thesaurus, we further implement a software-specific dictionary in a website[2] and a browser plugin[3] to help developers conveniently understand software-specific terms.
- We not only update previous experiments by including new data and new baselines, but also add more detailed analysis of experiments results.

1. Synsets are interlinked by means of conceptual-semantic and lexical relations
2. https://se-dictionary.appspot.com/
3. https://github.com/ccywch/se-dict-browser-extension

- To demonstrate the usefulness of the extracted thesauruses, we develop three thesaurus based applications, including software-specific text normalization, tag synonym detection in Stack Overflow and software-specific spell checker.
- To demonstrate the generality of our approach, we extend our approach to other kinds of Q&A sites including Mathematics Stack Exchange[4] and English Language & Usage Stack Exchange[5]. The experiment results demonstrate that our approach can also successfully construct domain-specific thesauruses from other data sources.
- To support the replication of this work, we also develop a python package[6] which implements our proposed approach. Developers or other researchers can adopt or customize it for their own purpose.

2 The Approach

The overview of our approach can be found in Fig. 2. The inputs of our approach are only a software-specific corpus (e.g., Stack Overflow text) and a general corpus (e.g., Wikipedia text). The output of our approach is a thesaurus including software-specific terms and their morphological forms (called SEthesaurus). Our approach mainly includes 6 main steps: 1) performing 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, such as FastText model [26] and continuous skip-gram model [27], 4) extracting semantically relevance terms as candidates of morphological forms, 5) discriminating abbreviations and synonyms from the list of morphological form candidates, and 6) grouping morphological forms of software-specific terms based on a graph of morphological relations.

2.1 The Inputs of Our Approach

The inputs of our approach are a software-specific corpus of plain text and a general corpus of plain text. No other external resources are required. In particular, software specific corpus can be crawled from domain-specific web sites, such as Stack Overflow, CodeProject, W3School, MSDN. As we are interested in discovering morphological forms from informal discussions in software engineering, as well as considering the popularity of the website and the volume of the data, we choose Stack Overflow text as the software-specific corpus in this work. General corpus can be crawled from domain-agnostic websites, such as Wikipedia, Quora, Baidu Zhidao, and these websites can cover a diverse set of domains. Considering the quality and the public availability of the data, we choose Wikipedia text as the general corpus in this work. Moreover, Wikipedia text is also adopted as the general corpus for other NLP work [28]. It is important to note that our data analysis approach is not limited to Stack Overflow and Wikipedia data.

Summary for the characteristics of the dataset: In this work, the Stack Overflow data dump[7], which is gathered from July 2008 to August 2018, contains 15,930,617 questions and 24,676,333 answers. We collect the title and body content of all the questions and answers as the software-specific corpus. The Wikipedia data dump[8] includes 8,556,773 articles before August 2018. We collect the page content of all the articles as the general corpus.

2.2 Performing Data Preprocessing for Input Corpuses

2.2.1 Text cleaning

Since both datasets are downloaded from websites, we perform the text cleaning steps, which are commonly used for preprocessing web contents [29], [30]. We first preserve textual contents but remove HTML tags. In particular, for Wikipedia data, we remove all references from page content. For Stack Overflow data, we remove long code snippets in <pre> or <code> in the posts, but not short code elements in <code> in natural language sentences. Then, we use our developed software-specific tokenizer [31] to tokenize these sentences. Our proposed tokenizer can preserve the integrity of code-like tokens and the sentence structure. For example, it treats $numpy.ndarray.argmin()$ as a single token, instead of a sequence of 7 tokens (i.e., {numpy, ., ndarray, ., argmin, (,)}).

2.2.2 Phrase Detection

A significant limitation of previous techniques for detecting similar words or synonym detection is that they only consider single word (i.e., token) [32]. However, many software-specific terms are composed of several words (such as *integrated development environment* , *operating system* and *breadth first search*). These multi-word phrases must be recognized and treated as a whole during data analysis.

We utilize a simple data-driven and memory-efficient approach [27] to recognize multi-word phrases in the text. In this approach, phrases are formed iteratively based on the unigram and bigram counts by using

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

w_i and w_{i+1} are two consecutive words, $count(w_i w_{i+1})$ returns the frequency of the phrase $w_i w_{i+1}$ in the corpus and N is the total number of words in the corpus. δ 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 δ times in the corpus. In our study, we set δ to 10 based on the previous study [27]. To determine the threshold of $score$, we carry out a small-scale pilot study (i.e., we ask the third author to manually check the coverage and accuracy of the detected phrases given different threshold values ranging from 5 to 20). The results show that setting the threshold of $score$ as 15 can achieve a relatively good balance between the coverage and accuracy.

Our method can find bigram phrases, which appear frequently enough in the text when 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

4. https://math.stackexchange.com/
5. https://english.stackexchange.com/
6. https://pypi.org/project/DomainThesaurus/
7. https://archive.org/details/stackexchange/
8. https://dumps.wikimedia.org/enwiki/latest/

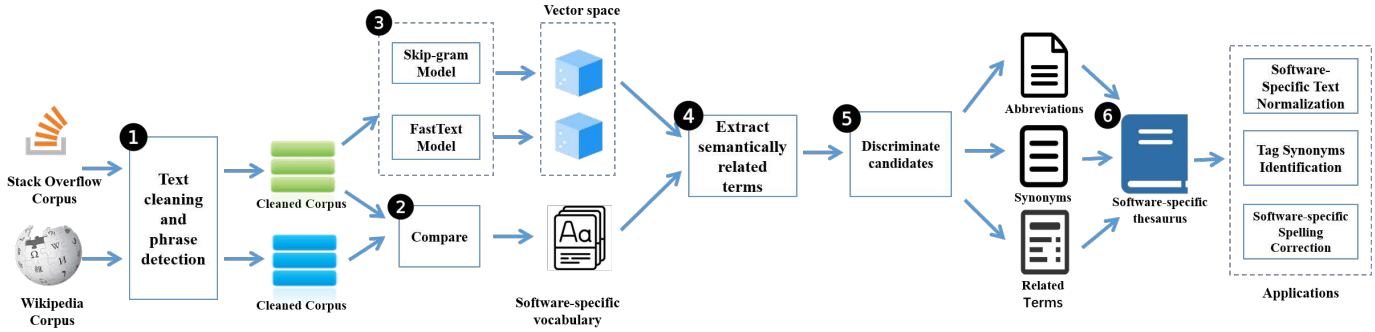


Fig. 2: The overview of our approach

the bigram phrases are formed, we repeat the process to detect trigram and fourgram phrases. In this work, we stop at fourgram phrases, but our approach can be extended to phrases with more words.

2.3 Building Software-Specific Vocabulary

Inspired by Park et al.'s work [28], we identify software-specific terms by contrasting the frequency of a term in the software specific corpus compared with its frequency in the general corpus. Specially, we measure a term's *domain specificity* by using 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 represent domain-specific and general corpus respectively, $p_d(t)$ and $p_g(t)$ are the probability of the term t in the two corpuses respectively. For example, the probability $p_d(t)$ of a term t in the software-specific corpus d is calculated by dividing the term frequency (i.e., $c_d(t)$) by the total number of tokens (i.e., N_d) in the corpus. The underlying intuition is that terms that appear frequently in the software-specific corpus but infrequently in the general corpus are software-specific terms. The detailed threshold of $domainspecificity$ is discussed in Section 4.6.

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

2.4 Learning Term Semantics

To learn the semantic of the term, we combine two approaches (i.e., the continuous skip-gram model [25], [27] and FastText model [26]). These models include state-of-the-art algorithms for learning distributed word vector representations (i.e., word embeddings) by using a neural network model, and they are widely used for software engineering domain [33], [34], [35]. The underling intuition of these algorithms is that words with similar meaning would appear in the similar context. Therefore, the representation of each word can be defined on the words it frequently co-occurs with.

The continuous skip-gram model regards each word as the basic unit, while the FastText model take the sub-word information into the consideration. Therefore, we complement the previous skip-gram model with FastText model so that our model can learn the term semantic from different granularities.

2.4.1 Continuous Skip-gram Model

As illustrated in Fig. 3, the objective of the continuous skip-gram model (skip-gram model) is to learn the word representation of each word, which 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:

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

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 2.2. We set the sliding window size N to 5 in our study. That is, the sliding window contains 5 surrounding terms as the context terms for a given term in the sentence. Notice the sliding window size is also an important parameter for the FastText Model and we also set the value of this parameter to 5. The sensitivity analysis for the value of this parameter can be found in Section 4.6.

The probability $p(w_{k+j}|w_k)$ in Equation 3 can be formulated as a log-linear softmax function:

$$p(w_{k+j}|w_k) = \frac{e^{s(w_k, w_{k+j})}}{\sum_{i=1}^{W} e^{s(w_k, i)}} \quad (4)$$

where W is the number of words in the vocabulary, and s is a scoring function to map pairs of words to scores for computation. In the continuous skip-gram model, the scoring function is the dot product of vectors of two words: $s(w_k, w_{k+j}) = {v_{w_k}}^T v_{w_{k+j}}$.

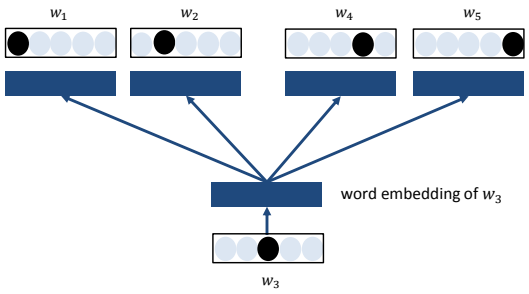


Fig. 3: The continuous skip-gram model, which can predict the surrounding words given the center word.

The Equation 4 can be efficiently solved by the negative sampling method [27]. 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.

2.4.2 FastText Model

Although skip-gram model can encode the semantic of software-specific terms into a dense vector, it only considers the word context rather than the word itself. For example, the term *javascript* and *javascrip* are a synonym pair, but as the misspelling *javascrip* appears rarely in the whole dataset, the vector learned from its context cannot fully represent its semantic. Therefore, it is possible that the vector of *javascript* and *javascrip* are not very close, but they have semantic similarity indeed. To overcome this problem, we take the sub-word information into consideration by adopting the FastText model [26].

Different from the skip-gram model, each word w is represented as a bag of character n-gram. For example, when we are considering character 3-gram (i.e., n is 3), the word *python* will be represented as a collection of n-gram:

$$< py,\ pyt,\ yth,\ tho,\ hon, on >$$

and also the special sequence $< python >$, where boundary symbols $<$ and $>$ annotate the beginning and the end of the word. Overall the FastText model follows the same target as that of the skip-gram model (i.e., Equation 3). But different from the skip-gram model, which considers the whole word as the unit, the scoring function in Equation 4 for the FastText model as the sum (i.e., dot product) of vector representations of its n-gram:

$$s(w_k, w_{k+j}) = \sum_{g \in G_{w_k}} {z_g}^T v_{w_{k+j}} \tag{5}$$

where $G_{w_k} \subset \{1, ..., G\}$ is the set of n-grams appearing in the given word w_k, and z_g is the vector representation associated with each n-gram g.

Apart from rendering high-quality vectors for rare words or terms, the FastText model can also give the representation for OOV (out-of-vocabulary) words since the OOV words can be also broken down into character n-grams. In contrast, the continuous skip-gram model cannot deal with the OOV words.

Note that when encoding the semantic of phrases, especially long phrases with several words such as **model-view-controller** and **drop-down-menu**, the FastText model behaves much worse than the skip-gram model. The reason for that problem is that the long phrases are decomposed into many sub-words, leading to the difficulty of encoding its semantic in the FastText model. That is why we combine both the skip-gram model and the FastText model for inferring semantic-related terms.

2.5 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 can 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:

$$\underset{w \in A_{V_i}}{\operatorname{argmax}} \cos(v(w), v(t)) = \underset{w \in A_{V_i}}{\operatorname{argmax}} \frac{v(w) \cdot v(t)}{\|v(w)\| \|v(t)\|} \tag{6}$$

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

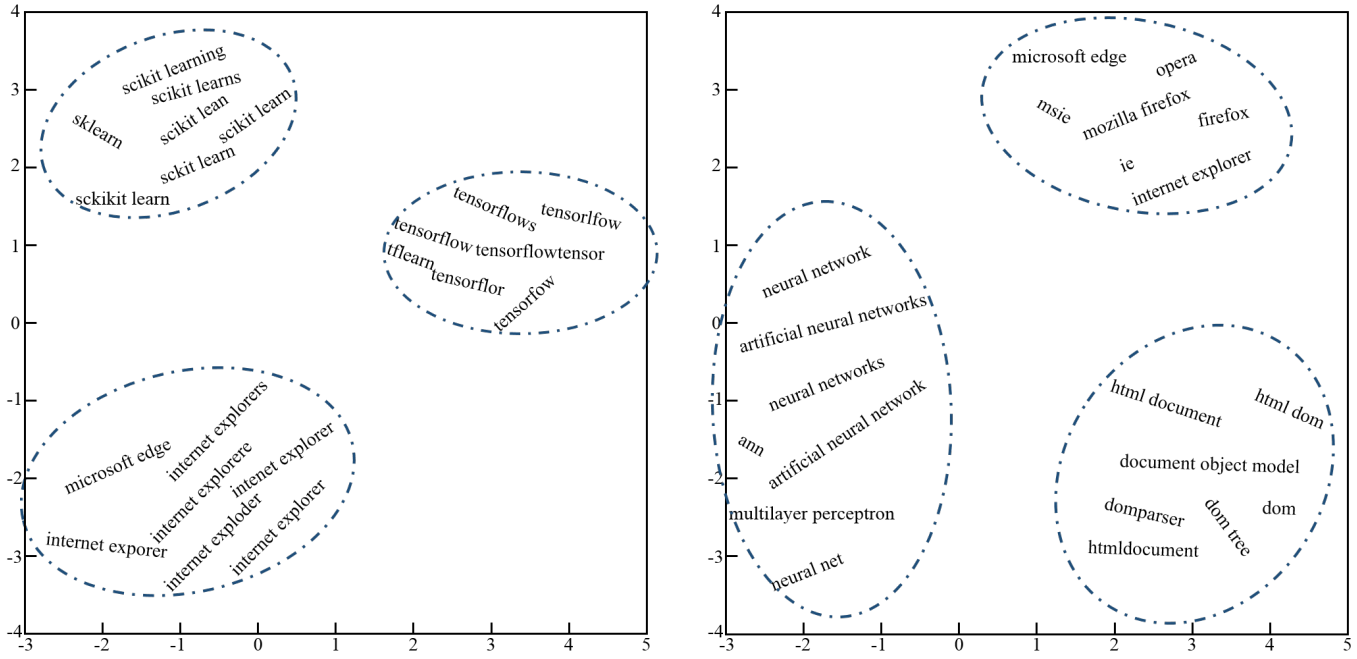
Given the skip-gram model (introduced in Section 2.4.1) and the FastText model (introduced in Section 2.4.2), for a term $t \in V_i$, we select the top-k (k=40) most similar terms in the vector space V_i based on the skip-gram model and the FastText model respectively. We use $Terms_S$ and $Terms_F$ to denote these two different term sets from the skip-gram model and the FastText model. Then we combine both sets ($Terms_S \cup Terms_F$) as the set of candidate semantically related terms. The sensitivity analysis for the value of the parameter k can be found in Section 4.6 To render a deeper understanding of word embedding methods, we visualize some terms in the vector space by adopting the PCA (Principal Component Analysis) method. Fig. 4 (a) shows the list of semantically related terms for the three terms (*scikit learning*, *tensorflow* and *internet explorer*) by using FastText model. Fig. 4 (b) shows the list of semantically related terms for the three terms (*artificial neural network*, *document object model* and *internet explorer*) by using continuous skip-gram model. Within the list of semantically related terms, there are three kinds of candidates: synonyms (i.e., morphological forms of a term in this work), abbreviations and other related words. Then we will introduce how to discriminate synonyms and abbreviations from the semantically related candidates for each term.

2.6 Discriminating Synonyms & Abbreviations

We now explain the lexical rules and the string edit distance, which are used to discriminate synonyms and abbreviations of a term from its semantically related terms.

2.6.1 Discriminating Synonyms

Synonyms are defined as pairs of morphological similar terms in this work. Some morphological-synonyms can be determined by using stemming, such as (*link, links*) and (*machine learn, machine learning*). However, many other synonyms can not be determined by this way, such as (*windows-service, windowsservices*), (*tf idf, tfidf*), (*integer,*



(a) FastText (b) continuous skip-gram

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

integet (a misspelling)), (*angular, angularjs*). We observe that morphological-synonyms among semantically related terms can be usually transformed from one term to another by performing 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 [36] is often used to compute the string edit distance. The string edit distance is the minimum number of single-character edits (such as insertions, deletions or substitutions) required to transform one word into another. In this work, we use Damerau-Levenshtein (DL) distance [37], which is an enhanced string edit distance, to compute the minimum number of single-character edits (such as insertions, deletions, substitutions, and transpositions) required to transform one term to another. DL distance enhances the Levenshtein distance [36] with the transposition of the two adjacent characters (such as *fa**l**se* and *f**la**se*). Such character transpositions are a common cause of misspellings. Therefore, DL distance can be used to detect such misspellings more reliably than Levenshtein distance.

The original DL distance can not be directly adopted for measurement. For example, the DL distance between a pair (*subdomain*, *sub-domains*) and the DL distance between a pair (*svg*, *png*) are both 2. However, the pair (*subdomain*, *sub-domains*) is morphological synonym, while the pair (*svg*, *png*) is not. Therefore, we take into consideration both the original absolute DL distance and the relative similarity between two terms. In this paper, the original DL distance of the two synonyms must not exceed 4. For example, the pair (*dispatcher.begininvoke, dispatcher.invoke*) will not be regarded as synonyms since the original DL distance between the two terms is 5.

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

$$similarity_{morph}(t, w) = 1 - \frac{DLdistance(t, w)}{max(len(t), len(w))} \quad (7)$$

where the function $DLdistance$ returns the original DL distance between two terms and the function len returns the length of the term. The relative similarity indicates that the different parts of the two synonyms should be relatively small when compared with the same parts of the two terms. In this work, we set the threshold for relative similarity as $1/3$ after carrying our a small-scale pilot study. As a result, the pair (*subdomain*, *sub-domains*) will be recognized as synonyms, but the pair (*svg*, *png*) will not. The reason is that the first pair is relatively similar enough, but the second pair is not.

2.6.2 Discriminating Abbreviations

Given a term t, if a semantically related term w does not satisfy the requirement of being a synonym, 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 they satisfy the following heuristics-based lexical rules. Similar lexical rules are used to identify abbreviations [15], [18].

- 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

TABLE 2: Representative terms and their abbreviations and synonyms.

Term	Abbreviations	Synonyms
access-control-list	acl	access-control-lists
windows-phone	wp	winphone
authentication	authn	autenticator, authentifications
database	db	databse, datbase
markdown	md	markdow, mardown
k-nearest-neighbour	knn	k-nearest-neighbor
button-click	btnclick, bt-click	buttonclicked, btnonclick

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-word 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 morphological synonyms and abbreviations are discriminated from highly semantically related terms established by the terms' word embedding. The above edit distance and lexical rules alone can not be used to detect morphological synonyms and abbreviations reliably without considering semantic relatedness between terms. For example, according to the above lexical rules, *ie* can be regarded as an abbreviation of *view*. 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 using the 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 a synonym candidate for each other.

2.7 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 term *files-system*, we obtain {*filessystem, filesysytem, file-system*}, while for the term *file-systems*, we obtain {*file-system, filesystems, file_systems*}. Note that the term *file-system* appears in two synonym sets simultaneously. 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 relation 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. Some examples[9] can be found in Fig. 5. For instance, the relationship between *files-system* and *file-systems* acts as a bridge between two synonym groups so that we can merge them as one.

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 for this 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 2 presents some examples of the representative terms and their abbreviations and synonyms identified by our approach.

9. For multi-word phrases (e.g., *multi modular*), we replace space with "_" for the convenience of visualization.

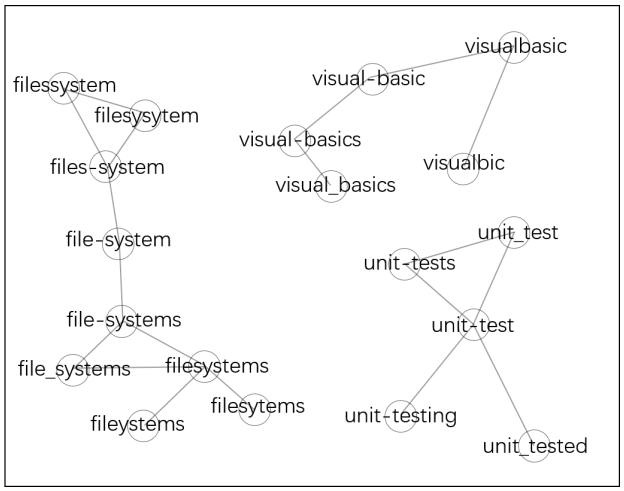


Fig. 5: The synonym graph (here the edge represents the synonym relationship and a connected component represents a group of synonyms)

TABLE 3: Exmaples of semantically related techniques

Term	Semantically related terms
java	scala, groovy, c++, clojure, c#, delphi, python
netbeans	eclipse, intellij, pydev, android stuio, aptana
beautifulsoup	lxml, nokogiri, html agility pack, jsoup, simplexml
codeigniter	cakephp, yii, zend, symfony, django, joomla, laravel
binary search	linear search, bubble sort, radix sort, quicksort

3 Implementation

Based on the approach mentioned in Section 2, we build a SEthesaurus with 466,228 software-specific terms, 18,951 abbreviations and 442,684 synonyms. Then we further enrich it with more information so that it can be regarded as a dictionary for software engineering domain. We consider all software-specific terms mined in our approach as entries in our dictionary. For each entry, there are four different kinds of information:

- Its abbreviations and synonyms;
- The definition from Wikipedia or TagWiki in Stack Overflow;
- Its semantic related terms, which are not included in the abbreviations and synonyms;
- Its frequent links in Stack Overflow, which are not included in Wikipedia or TagWiki in Stack Overflow.

The abbreviations and synonyms of each term come from our constructed SEthesaurus. We find its Wikipedia or Stack Overflow links according to the anchor text of the link. The semantic related terms are mined by the approach in Section 2.5. After extracting the synonyms and abbreviations, the rest is a list of terms, which are highly correlated with the given term. Table 3 shows some examples, and these terms can be exploited in recommendation systems

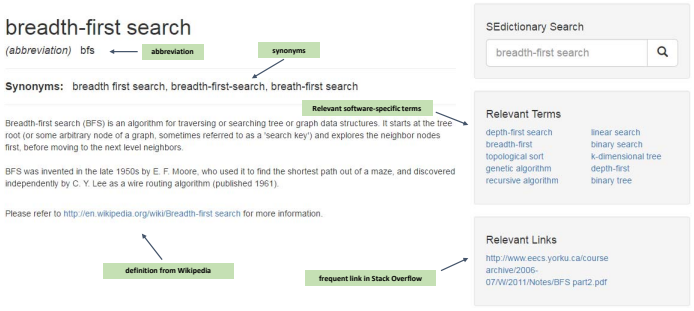


Fig. 6: The screenshot of our website

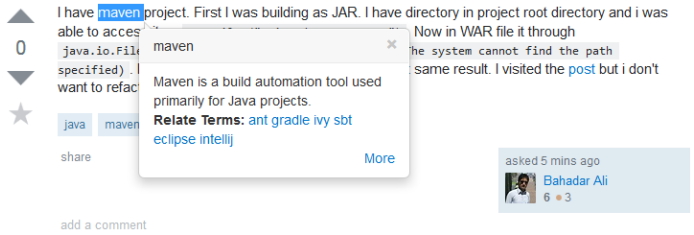


Fig. 7: The screenshot of the Firefox plug-in in use

for software engineering tasks. For example, we can use this knowledge to recommend similar techniques or analogical techniques across programming languages [38]. We may also exploit these semantically related terms for query expansion and reformulation [39], [40]. The frequent links are extracted in the same way as we obtain definition links from the Wikipedia or Stack Overflow.

We build a prototype demo website[10] for the community to have an access to our thesaurus and a screenshot of our website can be found in Fig. 6. In the website, users can search for software-specific terms and find their abbreviations and synonyms. As our thesaurus can also be applied to other related tasks, we also release the API (similar to WordNet API) for developers to have an access to the information in the dictionary. As an example, we develop a Firefox plug-in[11] based on our thesaurus, and this plug-in can assist users to understand software-related articles by showing the explanation for their interested software-specific terms. The screenshot of this plug-in can be found in Fig. 7.

4 Evaluation of Our Thesaurus

In this section, we evaluate the coverage of software-specific terms and the coverage of abbreviations and synonyms in our constructed thesaurus $SEthesaurus$ against several community-created lists of computing-related terms, abbreviations and synonyms. We also examine the correctness of the identified abbreviations and synonyms in the manual way, and demonstrate the generality of our approach by constructing thesaurus based other domain-specific data. We have released all experiment results in this study[12].

10. https://se-dictionary.appspot.com/
11. https://github.com/ccywch/se-dict-browser-extension
12. https://sedict.github.io/

TABLE 4: The coverage of software-specific terms in SOtag and CPtag datasets

Dataset	#Term	Method	#CoveredTerm	Coverage
SOtag	28335	SEthesaurus	17292	61.03%
		SEWordSim	2074	7.32%
		WordNet	4123	14.55%
CPtag	765	SEthesaurus	467	61.05%
		SEWordSim	341	44.58%
		WordNet	330	43.13%

4.1 Baselines

To demonstrate the effectiveness of our approach, we compare our SEthesaurus with two state-of-the-art baselines.

- **WordNet.** WordNet [23] is a general-purpose lexical database of English created by lexicographers. WordNet groups English words into synonym sets (synsets), such as {*small, little, minor*}.
- **SEWordSim.** SEWordSim [32] is a software-specific word similarity database, which is extracted from Stack Overflow.
- **SEthesaurus-sg.** SEthesaurus-sg [24] builds the SEthesaurus with solely continuous skip-gram model.
- **SEthesaurus-ft.** SEthesaurus-ft builds the SEthesaurus with solely FastText model.

4.2 The Coverage of Software-Specific Terms

To confirm whether our thesaurus can provide a good coverage ability for software-specific terms, we compare the software-specific terms of our thesaurus against two community-cumulated software-specific vocabularies from two popular programming Q&A sites (i.e., Stack Overflow and CodeProject).

In Stack Overflow, each question is attached with up to five tags, which are used to describe the programming techniques and concepts of this question. Therefore, these tags can be regarded as software-specific terms [41]. We collect 52,445 tags from 15,930,617 questions in Stack Overflow from July 2008 to June 2018. We only consider 28,335 tags which appear more than 30 times to avoid rare tags. Based on our observations, many rare tags are of low quality or not software-specific terms. For example, the tag *threads.h* is just a file in a library that supports multi-threading in c++ 11. Apart from Stack Overflow, we also collect 2,309 tags from 331,040 questions in CodeProject[13] from September 2005 to August 2018. Similar to Stack Overflow, we consider 765 tags whose frequency is larger than 30 times to avoid rare terms. Note that we take the same threshold on both CodeProject and Stack Overflow for the consistent experimental setting though the data quality of CodeProject is much lower than that of Stack Overflow [42], [43]. For the convenience of description, we refer these two community-cumulated software-specific vocabularies as *SOtag* and *CPtag*. The coverage of software-specific terms for our thesaurus and two baseline methods (i.e. WordNet and SEWordSim [14]) in SOtag and CPtag datasets can be found in Table 4. For each tag in

13. A popular programming Q&A site: http://www.codeproject.com/script/Answers/List.aspx??tab=active
14. The database from them is based on the Stack Overflow data by 2014.

the SOtag database or the CPtag database, we will check whether this tag is in the SEWordSim database provided by the authors and then calculate the coverage rate. In particular, our thesaurus contains 466,228 software-specific terms. It can cover 61.03% terms in the SOtag dataset and 61.05% terms in the CPtag dataset. In contrast, SEWordSim and WordNet can only cover no more than 50% terms. These comparison results demonstrate the necessity of building a software-specific thesaurus.

In Table 4, we notice the coverage of software-specific abbreviations for SEWordSim is low, especially in the SOtag dataset. One important reason is that SEWordSim only considers single words rather than multi-word phrases. According to our observation, there are 11,877 (41.92%) multi-word tags in the SOtag dataset and 63 (8.24%) multi-word tags in the CPtag dataset. That is one important reason why our model can significantly outperform the SEWordSim model on the SOtag dataset when compared to its performance on the CPtag dataset.

By further analyzing the tags in the SOtag and CPtag datasets, which are not covered by our thesaurus, we summarize three reasons and we use some tags to further illustrate these reasons. First, some tags in the SOtag and CPtag datasets contain phases with many words, and many of them contain version number (such as *google-play-services-3.3.0* and *facebook-graph-api-2.3*). However, developers often do not use version numbers when mentioning a technique in discussions. Therefore, our thesaurus may not contain a specific version of a technique, but it usually contains the general term for the technique, such as *google play services* and *facebook graph api*. As shown in the bar chart of Fig. 8, the coverage for the SOtags with 4 or more words is low (10.38%). However, for the tags with 3 or less words, the coverage becomes much higher. Second, as shown in Fig. 8, tags that are used more times, the coverage by our vocabulary is higher. In SOTags, for tags that are used more than 10000 times, the coverage by our thesaurus can reach 74%. But for tags that are used less than 100 times, the coverage is only about 57.09%. Note that, although the number of less-frequently-used tags (30-1000 times) is 11,507 and account for 40.61% of the tags in Stack Overflow, their total times of usage only account for 1.4% of the total tag usage. Therefore, the impact of missing some less-frequently-used tags for NLP tasks like information retrieval is trivial. Third, some tags are artificial terms for tagging the questions, such as *android-studio-import* and *azure-tablequery*, but these terms are rarely used in informal discussions.

4.3 The Coverage of Abbreviations

To confirm whether our thesaurus can provide a good coverage ability for abbreviations, we compare abbreviations of our thesaurus against a list of computing and IT abbreviations in Wikipedia[15]. This list contains 1,423 pairs of full name and corresponding abbreviation.

Our SEthesaurus mainly contains 18,951 abbreviations for 14,814 terms[16] from Stack Overflow corpus. 851 of these 1,423 full names in Wikipedia can be found in our thesaurus. For those 572 full names, which can not be found in our thesaurus, we summarize three reasons according to our observation. First, some of them are phrases with too many words (e.g., *national institute of standards and technology*, *java xml for remote procedure calls*). Second, some of them belong to other domains, such as communication domain (e.g., *atm adaptation layer, advanced research projects agency*). Third, some of them are just normal phrases, such as *Keyboard and personal computer*, which also frequently appear in the general text.

TABLE 5: The coverage of software-specific abbreviations

Method	#Covered FullName	#Covered Abbreviation	Coverage
WordNet	110	4	3.64%
SEWordSim	8	2	25.0%
SEthesaurus-sg	638	418	65.52%
SEthesaurus-ft	595	347	58.32%
SEthesaurus (top-40)	**721**	**511**	**70.87%**
SEthesaurus (top-20)	**721**	**487**	**67.55%**

With these 851 full names and their abbreviations in Wikipedia as the ground truth, we examine whether our SEthesaurus and baselines can cover them. The experimental results can be found in Table 5. Notice, in this table, SEthesaurus (top-40) denotes we select top-40 most similar terms from the skip-gram model and the FastText model and then combine these two sets as the candidate semantically related terms. While SEthesaurus (top-20) denotes we only select top-20 most similar terms from two different models. Considering SEthesaurus (top-20) can make the number of candidate semantically related terms of SEthesaurus at most 40. This setting can ensure a fair comparison with baseline methods, since these baseline methods only consider 40 similar terms. For SEthesaurus (top-20), the coverage rate of abbreviation of SEthesaurus is higher than SEthesaurus-sg and SEthesaurus-ft (the improvement is 2.03% and 9.23% respectively). This shows that combing two different models can effectively improve the coverage rate of abbreviation. For SEthesaurus (top-40), 721 (84.7%) of these 851 full names have abbreviations in our thesaurus, and our SEthesaurus covers the abbreviations of 511 (70.87%) full names in the ground truth. 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 are more likely to use full names instead of the abbreviations, e.g., *access method* instead of *am*, *access server* instead of *as*. Second, there are limitations in our abbreviation inference heuristics, which can not find abbreviations with unique string transformations, such as *802.11* for *wireless lan*, *gtk+* for *gimp toolkit*, and *i/f* for *interface*. In fact, our approach identifies these abbreviations as semantically related to their full names. However, due to their unique string transformation, general lexical rules can not be used to discriminate them as abbreviations. After compared with the Wikipedia abbreviation list, we can find that our thesaurus 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 find multiple abbreviations for a term. For example, our approach finds 6 abbreviations {*regex*, *reg exp*, *regexps*, *regexpes*, *regexp*, *regexs*} for *regular expression* in the Stack

15. https://en.wikipedia.org/wiki/List_of_computing_and_IT_abbreviations

16. Note that one term may have several abbreviations in our thesaurus.

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TSE.2019.2940439, IEEE Transactions on Software Engineering

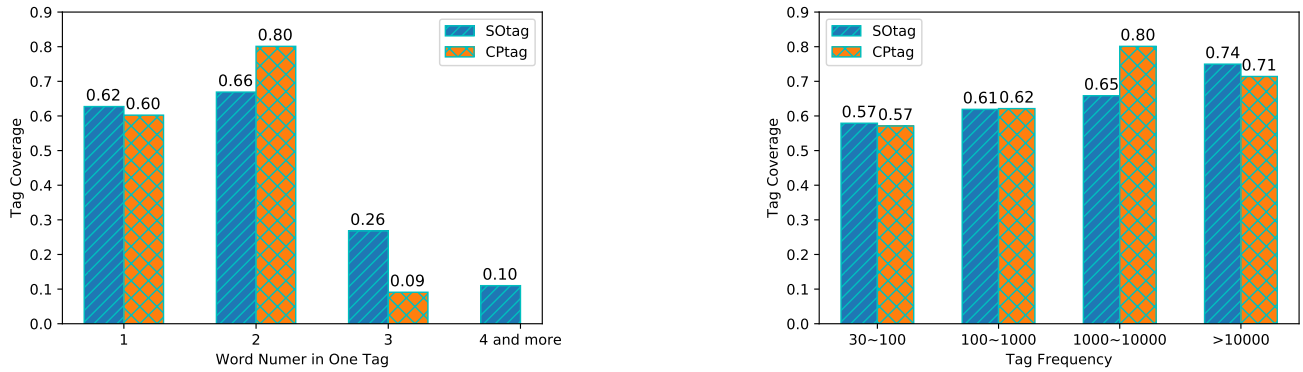


Fig. 8: The left bar chart shows the coverage rate with different number of words and the right bar chart shows that coverage varies with tag frequency.

Overflow text, while the Wikipedia list includes only two of these 6 abbreviations (i.e., {regex, regexp}).

4.4 The Coverage of Synonyms

Our approach identifies 60,844 synonym groups, which contain 442,684 morphological terms. To examine the coverage and accuracy of the identified synonyms in our thesaurus, we first compare our results against the tag synonyms in Stack Overflow. Stack Overflow community maintains a list of tag synonym pairs in the collaborative way. By August 2018, there are 3,663 community-approved synonym pairs[17]. Each pair has a synonym tag and a master tag. We take these tag synonym pairs as the ground truth. According to the tag naming convention in Stack Overflow, multi-words in a tag is concatenated by "-". But in the plain text, users are more inclined to use spaces to separate them. Thus, we replace "-" in the tag with space for the fair comparison. For example, the tag *visual-studio* will be transformed into *visual studio*.

Then we compare our approach with the four baseline methods (i.e., WordNet, SEWordSim, SEthesaurus-sg and SEthesaurus-ft). When using WordNet as the baseline, for each synonym tag (e.g., *videojs*) in the ground truth, we check whether it is in a synonym group that WordNet identifies, and if so, we further check whether its corresponding master tag (e.g., *video.js*) is also in the synonym group. When using SEWordSim as the baseline, for each synonym tag in the ground truth, we check whether it is in the SEWordSim database, and if so, we further check whether the master tag is in the list of the top-40 most similar words for the synonym tag in the SEWordSim database. Notice SEWordSim provides an API to check whether a term is in their database and can return this term's top-k most similar words. When using SEthesaurus-sg, SEthesaurus-ft and SETheasurus, for each synonym tag in the ground truth, we check whether it is in a synonym group, and if so, we further check whether its corresponding master tag is also in the synonym group.

The experimental results can be found in Table 6. Consistent with Section 4.3, we also show the results of SEthesaurus (top-40) and SEthesaurus (top-20) in this table. For SEthesaurus (top-20), the coverage rate of synonyms of SEthesaurus is also higher than SEthesaurus-sg and SEthesaurus-ft (the improvement is 6.78% and 2.57%

17. http://stackoverflow.com/tags/synonyms

TABLE 6: The synonyms coverage of our thesaurus

Method	#Covered Synonym	#Covered Master	Coverage
WordNet	828	310	37.44%
SEWordSim	381	42	11.02%
SEthesaurus-sg	2,718	1,610	59.23%
SEthesaurus-ft	2,987	1,895	63.44%
SEthesaurus (top-40)	**2,989**	**2,125**	**71.09%**
SEthesaurus (top-20)	**2,989**	**1,973**	**66.01%**

respectively). This verifies that combing two different models can effectively improve the coverage rate of synonym. For SEthesaurus (top-40), our SEthesaurus can cover 2,989 (81.60%) out of 3,663 synonym tags in the ground truth, while WordNet and SEWordSim can only cover 828 (22.6%) and 381 (10.4%) respectively. By comparing SEthesaurus-sg and SEthesaurus-ft, we can find that FastText model contributes more to our SEthesaurus than skip-gram model, in term of synonym detection. By further analyzing 2,989 synonym tags, 2125 (71.09%) correct synonyms are contained in our synonym groups.

We further analyze the reasons why our approach misses 864 synonym pairs. First, some synonym pairs are not morphological and thus they are beyond the ability of our approach, such as (*cgridview, yii*) and (*dcos, mesosphere*). Second, the Stack Overflow community sometimes merges fine-grained concepts into more general ones as tag synonyms, such as (*custom font, font*), (*twitter oauth, twitter*) and (*memorystream, stream*). However, such fine-grained terms and general terms have different meanings in the discussion text, and our approach does not regard them as synonyms. Last, some synonym pairs are not semantic related, such as (*arraycopy, arrays*) and (*digest, hash*).

4.5 Human Evaluation for Our Constructed Thesaurus

Based on the above evaluations, compared with several community-cumulated ground truth, our thesaurus contains much more software-specific terms, and a term in our thesaurus often has several abbreviations and synonyms. However, our evaluations against these community-cumulated ground truth only show that the correctness of a subset of abbreviations and synonyms identified by our approach. Moreover, these evaluations do not show whether many other abbreviations and synonyms, which are not included in the ground truth are correct or not.

To verify the correctness of the abbreviations and synonyms in our thesaurus, we recruit 2 master students majoring in computer science to perform the manual evaluation. They both have several-year programming experience and the quality of human evaluation can be guaranteed. As we have a large number of abbreviations and synonyms to examine, we adopt a sampling method to select the validation set. According to the widely-used sampling method [44], we examine the minimum number MIN of data instances to ensure that the estimated population is in a certain confidence interval at a certain confidence level. This MIN can be determined by the formula:

$$MIN = \frac{n_0}{1 + \frac{n_0 - 1}{populationsize}} \quad (8)$$

here n_0 depends on the selected confidence level and the desired error margin: $n_0 = (Z^2 \times 0.25)/e^2$, where Z is a confidence level z score and e is the error margin. For the final human evaluation, we examine MIN instances of relevant data for the error margin $e = 0.05$ at 95% confidence level (i.e., $MIN = 384$).

Therefore, we randomly sample 384 abbreviation pairs and 384 synonym pairs in our thesaurus for the human evaluation. For each sampled pair, these two students first independently evaluate its accuracy. Then, we compute Cohen Kappa [45] to evaluate the inter-rater agreement. For the pairs that the two students disagree, they have to discuss and come to final decisions. Based on the final decisions, we compute the accuracy of mined abbreviations and synonyms. Note that to check if there are any human evaluation bias, we carry out one more experiment (i.e., inserting 50 incorrect pairs among 50 randomly sampled pairs from our method, without telling the participants). The results show that two participants can accurately identify all those 50 wrong pairs, demonstrating the fairness of our method.

The human evaluation confirms that 280 (72.9%) abbreviation pairs and 332 (86.5%) synonym pairs[18] are correct. The Cohen Kappa metric among annotator decisions is 0.925 and 0.937 for abbreviations and synonyms, which indicates almost perfect agreement [46].

We further analyze the reasons for those incorrect pairs. In particular, two reasons result in the incorrect abbreviation pairs. First, the rules described in Section 2.6 could erroneously classify terms as abbreviations, such as *windows xp* as the abbreviation of *windows explorer*, or *waver* as the abbreviation of *waveform*. 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*. There are also two reasons resulting in the incorrect synonyms. First, most incorrect synonym pairs are caused by term pairs, which are both semantically and lexically similar, but are not synonyms, such as (*windows, windowsrt*) and (*openssh, openssl*). Second, other incorrect synonym pairs are also caused by erroneous synonyms and synonym grouping, which are similar to the example of the incorrect abbreviation pairs, such as (*btle, bluetooth*).

18. Detailed experimental results are released in https://sedict.github.io/

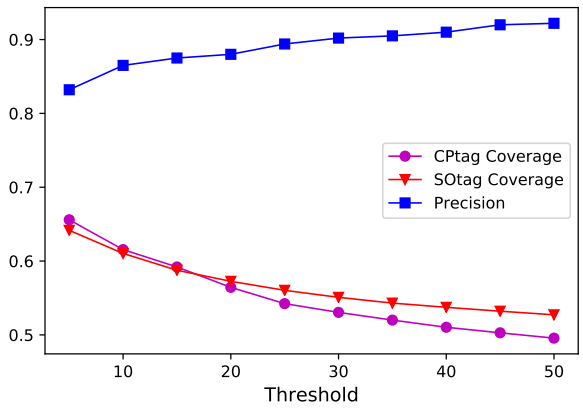


Fig. 9: Precision and coverage rate with different threshold values

4.6 Analysis on Parameter Optimization

There are some parameters needing optimization in our proposed unsupervised approach. In this subsection, we mainly perform sensitivity analysis for three parameters (i.e., domain specificity score, sliding window size, and top-k similar terms).

4.6.1 Domain Specificity Score

The threshold for $domainspecificity$ can be used to discriminate software-specific terms and the details of this parameter can be found in Section 2.3. We try different values within the range from 5 to 50 with the step as 5. For each value, we run our program to obtain all domain-specific terms and sample a fixed number of terms by adopting Equation (8) for manual evaluation. We recruit 2 master students majoring in computer science to perform the manual evaluation. For each sampled term, the two students independently evaluate its accuracy. For the terms that these two students disagree on, they have to discuss and come to final decisions. When computing coverage rate, we use tags in Stack Overflow and CodeProject as the ground truth to compute the coverage rate. The final results can be found in Figure 9. As seen in the figure, if the threshold is larger than 10, the precision rate will not be improved significantly, while the coverage rate will decrease significantly. Therefore, we set the value of threshold to 10 in our study for balancing the precision and coverage.

4.6.2 Sliding Window Size

The sliding window size is an important parameter for both the continuous skip-gram model and the FastText model. We carry out an experiment to perform sensitivity analysis on this parameter. In particular, we sample the sliding window size from 1 to 30 with step size as 1 to the skip-gram model and the FastText model respectively for generating corresponding thesauruses. First we calculate the coverage

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TSE.2019.2940439, IEEE Transactions on Software Engineering

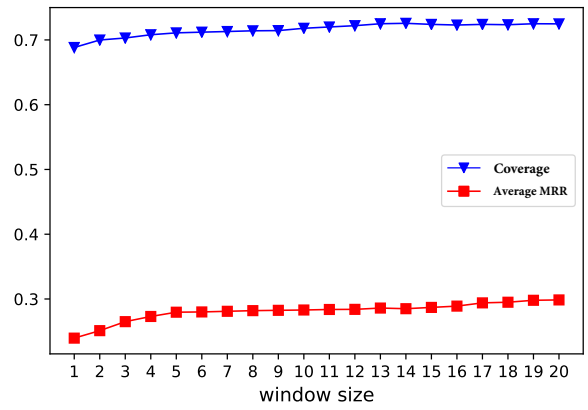


Fig. 10: Coverage and MRR of synonym tag with different values of the sliding window size

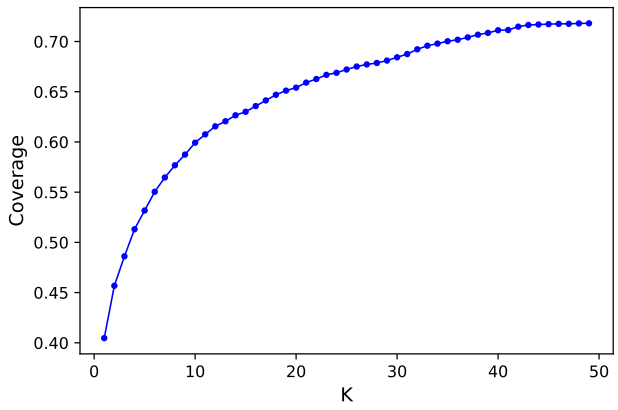


Fig. 11: Coverage rate with different values of the parameter k

of synonyms (i.e., #Covered Master / #Covered Synonym), which is introduced in Section 4.4. We also compute MRR (Mean Reciprocal Rank) [47] to evaluate the precision of the constructed thesauruses. Let k be the rank position of the first morphological form in the candidate list for a query term, then the reciprocal rank (RR) is defined as $\frac{1}{k}$. MRR is the mean of the RRs over all queries. For each pair in our ground truth (introduced in Section 4.4), we get the master tag's rank among the synonym tag's semantic-related words. As there are two models (i.e., the skip-gram model and the FastText model), we take the mean value of the two ranks as the final rank. We compute reciprocal rank for all pairs and take the mean value as the thesaurus' MRR.

The results can be found in Figure 10 and show that with the increase of window size, the coverage and MRR value are both increasing. But note that the increase is not significant, but relatively steady especially when the window size is larger than 5. However, the larger the window size, the longer the training time. Therefore, we set the value of this parameter to 5 for balancing the model effectiveness and training efficiency.

4.6.3 Top-k Similar Terms

Top-k similar terms is an important parameter when extracting semantically related terms. To analyze the influence of this parameter, we test its value from 1 to 50 with the step size as 1. For each value, we check the coverage of synonyms as mentioned in Section 4.4 which shows the ability of our model in finding semantic related terms. Figure 11 shows the coverage rate with different values of the parameter k. In this figure, we can find that with the increase of the value k, the coverage is also increasing. But when the value is larger than 40, the improvement is not significant. Therefore, we set k as 40 in our study.

Notice that the MRR of synonym tag with different values in Figure 10 is the average value of both the continuous skip-gram model (i.e., SEthesaurus-sg) and the FastText model (i.e., SEthesaurus-ft). While in Figure 11, the coverage rate of synonym tag with different values of the parameter k is based on our proposed SEthesaurus. Supposing that there are 10 query terms, we get the correct synonyms in the 4 of them in the SEthesaurus-ft model but 0 of them in the SEthesaurus-sg model, leading to average MRR as 0.2. When the same queries are applied for top-1 coverage, it is based on the combination of SEthesaurus-ft and SEthesaurus-sg (i.e., 4 of them are hit), leading to 0.4. That is why average MRR can be smaller than top-1 coverage in these two figures.

4.7 The Generality of Approach

To demonstrate the generality of our approach, we select two other sites in other domains from Stack Exchange Networks (i.e., Mathematics Stack Exchange and English Language & Usage Stack Exchange. Different from Stack Overflow, which is more related to engineering (i.e., software engineering), the Math site is more related to science, and the English usage site is more related to linguistics. By applying our approach to these sites, we successfully construct the thesaurus for each of them. For mathematics thesaurus, our approach mines 89,757 domain-specific terms, 1,624 abbreviations and 81,572 synonyms. For English-language thesaurus, our approach extracts 99,285 domain-specific terms, 370 abbreviations and 8,519 synonyms.

We repeat the same evaluation process to evaluate these newly constructed thesauruses. For mathematics thesaurus, it covers 867 (66.08%) tags in its site. For English-Language thesaurus, it covers 267 (58.81%) tags in its site. One Wikipedia page[19] summarizes 261 pairs of mathematical abbreviaiton pairs. Within 261 pairs, our mathematics thesaurus covers 118 full names and 78 (66.10%) abbreviations.

Similar to synonym pairs in Stack Overflow, the community Mathematics Stack Exchange[20] and English Language & Usage Stack Exchange[21] also maintain their synonym pairs. There are 157 synonym pairs in the math site and 229 synonym pairs in English site. By taking them as the ground truth, our mathematics thesaurus covers 111 (70.70%) synonym tags. For these synonym tags, it covers 81 (72.97%)

19. https://en.wikipedia.org/wiki/List_of_mathematical_abbreviations
20. https://math.stackexchange.com/tags/synonyms
21. https://english.stackexchange.com/tags/synonyms

master tags. Our English-language thesaurus covers 191 (83.41%) synonym tags. For these synonym tags, it covers 137 (71.73%) master tags. More detailed experiment results analysis can be found in Appendix. These results show that given a corpus of domain-specific data, our approach can be used to automatically construct a domain-specific thesaurus.

5 Usefulness of Our Thesaurus

To demonstrate the usefulness of the constructed thesaurus, we develop three applications based on our thesaurus in this section. We not only describe how we develop these applications, but also evaluate the performance of these applications by comparing with state-of-the-art baselines.

5.1 Supporting Software-Specific Text Normalization

The first application is to assist software-specific information retrieval by using our thesaurus.

5.1.1 Background

NLP-based techniques have been widely used to support software engineering tasks involving text data [48], [49], [50]. 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 [15], [16]. Abbreviations and synonyms are often referred to as inflected (or derived) words in natural language processing. The goal of text normalization is to transform inflected (or derived) words to their root form. Techniques for general English text (such as stemming [51] or WordNet lemmatization) are commonly adopted for software engineering text. Some studies propose domain-specific techniques to normalize source code vocabulary (e.g., expanding abbreviated identifiers). However, none of previous studies examine the normalization of informal software engineering text on social platforms.

5.1.2 Experiment Setup

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[22], which is another popular Q&A web site for computer programming. We perform data preprocessing for these sampled questions in the same way, which is described in Section 2.2.

Baseline Methods. The task is to normalize the title and content of the sampled questions. We develop a software-specific lemmatizer based on our thesaurus for normalizing abbreviations and synonyms in informal software engineering text. We compare the performance of our developed lemmatizer with two baseline methods, which are commonly used for text normalization. These two baseline methods are Porter stemming [51] and WordNet-based lemmatization. In particular, for our developed lemmatizer, we transform 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 transforms different forms of a word to their lemma based on WordNet synset (i.e., set of synonyms created by highly trained linguists).

22. http://www.codeproject.com/script/Answers/List.aspx??tab=active

Fig. 12: The tags and their different forms in the question title and content

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 way as we normalize the title and content of the question using the specific text normalization method. To mimic the information retrieval in software-engineering context i.e., we put the tags of posts as the query, and the question body as the documents. After the normalization, the more common words the query and question body share, the higher chance the question body can be searched by the tag query. Then, we measure the effectiveness of a text normalization method by coverage percentage of tags in the title and content of the question 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. 12 takes a question as an example. Before text normalization, none of these two tags appears in the title and content of the question. After using our lemmatizer for normalization, all of these two tags appear in the title and content of the question.

5.1.3 Result Analysis

As shown in Fig. 13, without text normalization, on average only 52.61% and 33.82% tags appear in the title and content of the sampled questions from Stack Overflow and CodeProject, respectively. This indicates that the consistency between question texts and question metadata is low. After performing text normalization by our lemmatizer, the percentage of tag coverage is boosted to 74.85% for the sampled questions from Stack Overflow questions, and 59.48% for the sampled questions from CodeProject questions. Although Porter stemming and WordNet-based lemmatization can also improve the consistency between question texts and question metadata, the improvement in the percentage of tag coverage is much smaller than our lemmatizer.

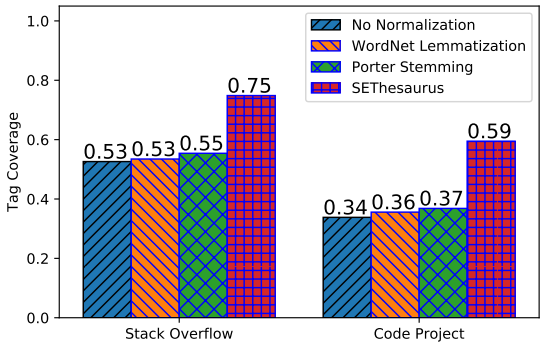


Fig. 13: The average percentage of tags appearing in question text by different text normalization methods

The Porter stemming can only find terms with derivational affixes, such as (*upload, uploaded*) or singular and plural forms, 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 constructed by mining the vast amount of software engineering text. Therefore, it can contain a much richer set of software-specific terms, including their abbreviations and synonyms. Furthermore, our thesaurus can recognize complicated synonyms, such as (*angular.js, angularjs*) and (*inheirt, inherit*). However, these complicated synonyms are difficult to find by using Porter stemming and WordNet lemmatization. In summary, our domain-specific thesaurus is more suitable for software-specific text normalization than general stemming methods or general English thesaurus.

5.2 Tag Synonyms Identification on Stack Overflow

There are tens of thousands of tags in Stack Overflow used by different users in the community. Due to the diversity of the human language, it is very likely that same-meaning tags with slightly different forms co-exist in the site (e.g., *pdfjs*[23], *pdf.js*[24]). Such synonym tags may cause the confusion for users and make it difficult to retrieve posts by tags.

Although the community has proposed a list of synonym tags[25], there are still many more synonyms, which are not discovered. In this section, we directly apply our approach to find tag synonyms to complement existing synonym list cumulated by the community manually. Note we need to mine synonym tags, therefore we change the dataset from text to all tags attached to questions in Stack Overflow.

In Stack Overflow, there are 15,930,617 questions and each question must be attached with up to 5 tags. Therefore, we regard tags for each question as a sentence, and feed them into word embedding model. More details can be found in our previous work [38]. For each tag, given its semantic words from word embedding, we further exploit the rule-based methods proposed in this paper to identify its abbreviations and synonyms. Finally, we extract 758 pairs of synonyms and 158 pairs of abbreviations.

We paste all pairs into a post[26] in Meta Stack Overflow, which is a site for users to discuss the workings and policies of Stack Overflow rather than discussing programming itself. Until now, it has received 76 upvotes, 9 favorite question marks and some appreciating comments. The community have checked that 290 of them are synonyms or abbreviations, 22 pairs are not synonyms or abbreviations, and leaving the other 604 as undecided. The posts have been viewed more than 1000 times so far. It shows that our approach can help to avoid the appearance of duplicated tags and it can help to maintain the quality of Stack Overflow.

23. http://stackoverflow.com/tags/pdfjs/info
24. http://stackoverflow.com/tags/pdf.js/info
25. http://stackoverflow.com/tags/synonyms

TABLE 7: Misspelling examples in our thesaurus

Term	Misspellings
entity-framework	enitityframework, entityframewrok
wordpress	wordpressy, wordprss, wordppress
mongodb	mogodb, monogd, mondgodb, monglodb
variable	varriable, varible, vairable, varaiables, varibles
servlets	servle, servelt, servlett, servet

5.3 Software-specific Spelling Correction

5.3.1 Background

When developers write natural-language documents, such as comments, documentations, blogs or Q&A posts, it is natural that misspellings often occur. Stack Overflow is the most popular programming Q&A site partially because of the community efforts to maintain the quality of the questions and answers in the website [52]. In Stack Overflow, the low-quality questions and answers will be edited by the senior users to guarantee their quality for other readers. For some editions, there are also some comments to briefly describe what they change. Among these editions, 340,120 of them are about the spelling problem, since the word "spell" appears in these comments. Note that this number is highly underestimated, as not every editor will mark their editions with detailed comments. Some misspelling examples in our thesaurus can be found in Table 7. To avoid these misspellings, developers need tools to support spell checking in their editors or IDEs. Some researchers [53], [54] use spell checkers (e.g., Aspell[27], Hunspell[28]) to preprocess software engineering text. However, these spell checkers are trained on general English text and do not consider software-specific terms and their common misspellings which rarely appear in the general text. Therefore, there is an urgent need to develop a software-specific spelling checking tool to alleviate the community efforts so that they can focus on more important editions.

5.3.2 Experiment setup

Collecting dataset: We first get all editing history of question titles in Stack Overflow[29]. Then, we compare the edition and the original post, note that only the pairs which are

26. http://meta.stackoverflow.com/questions/342097/a-list-of-tag-synonyms-not-proposed-in-stack-overflow
27. http://aspell.net/
28. https://hunspell.github.io/
29. https://archive.org/download/stackexchange/stackoverflow.com-PostHistory.7z

TABLE 8: Performance comparison for spelling correction

Method	# Covered Misspellings	Coverage	# Accurate Corrections	Accuracy
Hunspell	572	73.05%	500	87.41%
Aspell	576	73.56%	499	86.63%
SEThesaurus	638	81.48%	522	81.82%
Hunspell + SEThesaurus	703	89.78%	**641**	**91.18**%
Aspell + SEThesaurus	**709**	**90.55%**	639	90.13%

similar enough (i.e., Jaccard similarity is larger than 0.8) are left. After that, we use the differencing tool[30] to discriminate the replacement of words between the original title and after-edit title. We store all of these editing pairs (i.e., (*original word*, *replacement word*)). Then, we adopt the length difference to filter out some obvious errors, which are not misspellings, such as pairs (*please help to understand*, *understanding*). Later, we take the replacement pairs, whose frequency is larger than 10, to avoid the noises. The auto-correction method of SEthesaurus is based on the historical knowledge. Therefore, we partition the data into two parts according to the creation time. In particular, the data between 2008 and 2015 is used to generate SEthesaurus and the remaining data (i.e., the data between 2016 and 2018) is used to extract ground truth. Finally, we collect 783 replacement pairs as the testing dataset.

Baselines: We take two widely-used spelling checker (i.e, Hunsepll and Aspell) for comparison with our spelling checker (i.e., SEdic). For each pair, we firstly check whether the original word can be judged as misspellings by these two methods. Once the one is judged as a wrong spelling, we mark it as covered. Then, we further check whether the edited ones in the ground truth are also in the suggestion list for the given word by these two methods. Finally, we calculate the accuracy for all pairs.

Our auto-correction method based on SEthesaurus is slightly different from two baseline methods. First, the SEthesaurus is stored as a thesaurus by hashmap with the normalized form as the key, and all its synonyms and abbreviations as the value. We mark the pair as covered if the original word is in our dataset. Then for the covered pairs, if the normalized form of that original word is the replacement word in the ground truth, we mark it as accurate.

5.3.3 Result Analysis

The results can be found in Table 8. From this table, we can find that the coverage rate of our SEThesaurus based spell checker is higher than Hunspell and Aspell, although many misspellings in Stack Overflow also widely appear in the general text, such as *(anther, another)*, *(shure, sure)*. That is because many misspellings are derived from software-specific terms, which can not be covered by the general spelling checkers. To further take both the general spelling error and domain-specific typo into the consideration, we combine the general spell checker with the specific SEthesaurus to process software-specific text. Table 8 shows that both the coverage rate ($> 85\%$) and accuracy ($> 90\%$) are much higher than any single method.

6 Threats to Validity

In this section, we mainly discuss the potential threats to validity of our empirical studies.

Threats to internal validity are mainly concerned with the uncontrolled internal factors that might have influence on the experimental results. The first threat is the potential faults introduced during our approach implementation. To reduce this threat, we use manually designed test cases to verify the correctness of our implementation. We also use some mature third-party libraries to implement our approach. For example, we use the implementations for the skip-gram model and the FastText model provided by the gensim package[31]. Besides, there exists many parameters in our approach and inappropriate parameter values will affect the quality of constructed thesaurus. To select appropriate values for these parameters, we perform sensitivity analysis and the results of sensitivity analysis for these parameters can be found in Section 4.6.

Threats to external validity are about whether the observed experimental results can be generalized to other subjects. Based on the above analysis, we find our proposed approach can automatically construct software-specific thesaurus. To demonstrate the generality of our approach, we select two sites in other domains from Stack Exchange Networks (i.e., Mathematics Stack Exchange and English Language & Usage Stack Exchange) and build thesauruses for these two domains. Then, we also design some experiments (i.e., tag coverage, abbreviation coverage and tag synonyms coverage) and the results show that our approach still has a good performance in other domain corpora.

Threats to conclusion validity are mainly concerned with judging bias in human evaluation (in Section 4.5). To ensure that these two students can judge the pairs fairly and objectively, we design additional experiments in which we randomly extract 50 wrong pairs for 50 times independently. Then we let these two students verify the correctness of extracted pairs and find that the students can correctly identify all the wrong and correct pairs.

7 Related Work

During the process of software development, developers often use abbreviations and acronyms in identifiers and domain-specific terms in source code and documentation of the project. This phenomena pose challenges to the effectiveness of NLP-based techniques in exploiting software text. Previous studies attempting at expanding abbreviations in identifiers often resort to string edit distance [55], string frequencies from source code [56], [57], word co-occurrence data [18], or a combination of several techniques. These approaches inspire the design of lexical rules and transformations in our approach.

Informal discussions on social platforms for developers (e.g., Stack Overflow, CodeProject) contain many abbreviations, synonyms and misspellings. Furthermore, informal discussions on social platforms cover a much broader range of programming knowledge, when compared with traditional software text that previous work has focused on. These facts could make heuristics developed for expanding

30. https://docs.python.org/2/library/difflib.html

31. https://radimrehurek.com/gensim/

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 relevance between software-specific terms and determine morphological forms. One way is to develop a software-specific thesaurus like the WordNet [23] for general text, and the BioThesaurus [58] for biology domain. In fact, most of identifier expansion approaches [15], [16], [17], [18], [19] use this thesaurus or the other external dictionary (such as general English dictionary, domain-specific ontology and/or a list of well-known abbreviations) and then rank the relevance of abbreviation and expansions. Although such dictionaries are useful for reasoning word similarity, 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 dictionaries. Indeed, our study presents an automatic approach to construct a high-quality thesaurus of software-specific terms and morphological forms by mining the text data alone.

Researchers have made some attempts to automatically infer semantically related terms in software engineering text [18], [20], [21], [22], [59], [60]. However, some important assumptions are made on these proposed techniques, for example, relying on project-specific naming or documentation convention [22], [61], lexical difference patterns between sentences [21], or the availability of certain contextual information (e.g., dictionary, question tags) [18], [32], [62]. Such assumptions lack the generality for other data. Different from previous studies, we resort to unsupervised word representations (i.e., word embeddings) [27] 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 [1], [63], API/library recommendation [38], [64], [65], [66], semantically related posts prediction [6]. Different from previous studies, based on semantically similar software-specific terms, we want to infer a group of morphological forms which represent a distinct concept in informal discussions.

Beyer et al. [67] 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. Then, they [68] use community detection techniques, which can assist in the analysis of topic trend on Stack Overflow, to group tag synonym pairs into topics. 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 more than ten-times software-specific terms and discovers morphological relations among these terms; 2) their work only studies Stack Overflow tags, while our work applies the morphological forms, which are mined from Stack Overflow, to normalize the text from CodeProject.

8 Conclusion 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, and we also successfully verify the generality of our approach in constructing domain-specific thesaurus given other datasets. Based on the extracted thesaurus from Stack Overflow, we further develop three applications to demonstrate the usefulness of the thesaurus.

In the future, we want to deepen our research in two directions. First, we will improve the effectiveness of our method by refining each step in the pipeline including adopting better phrase-detection algorithm, filtering out more false positives. As there are many ad-hoc rules in this work, we hope to adopt some supervised learning methods to fully automatically extract those rules for extracting morphological terms. On the other hand, we will also extend our study into other domains for constructing domain-specific thesaurus like medicine or social science.

Acknowledgement

This work is partially supported by the seed grant from Monash University, National Natural Science Foundation of China (Grant Nos. 61702041, 61602267, 61872263, 61202006), Jiangsu Government Scholarship for Overseas Studies. Xiang Chen and Chunyang Chen have contributed equally for this work and they are co-first authors. Chunyang Chen is the corresponding author.

References

[1] 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*. ACM, 2016, pp. 404–415.

[2] Y. Zou, T. Ye, Y. Lu, J. Mylopoulos, and L. Zhang, "Learning to rank for question-oriented software text retrieval," in *Proceedings of 2015 30th IEEE/ACM International Conference on Automated Software Engineering*. IEEE, 2015, pp. 1–11.

[3] M. Nassif, C. Treude, and M. Robillard, "Automatically categorizing software technologies," *IEEE Transactions on Software Engineering*, pp. 1–1, 2018.

[4] D. Ye, Z. Xing, C. Y. Foo, J. Li, and N. Kapre, "Learning to extract api mentions from informal natural language discussions," in *Proceedings of 2016 IEEE 23rd International Conference on Software Analysis, Evolution, and Reengineering*. IEEE, 2016, pp. 90–101.

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

[6] 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 *Proceedings of 31st IEEE/ACM International Conference on Automated Software Engineering*. ACM, 2016, pp. 51–62.

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

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

[9] 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*. ACM, 2016, p. 14.

[10] C. Chen, Z. Xing, and L. Han, "Techland: Assisting technology landscape inquiries with insights from stack overflow," in *Proceedings of the 2016 IEEE International Conference on Software Maintenance and Evolution,*. IEEE, 2016, pp. 356–366.

[11] M. Creutz and K. Lagus, *Unsupervised morpheme segmentation and morphology induction from text corpora using Morfessor 1.0*. Helsinki University of Technology Helsinki, 2005.

[12] J. A. Botha and P. Blunsom, "Compositional morphology for word representations and language modelling." in *Proceedings of the International Conference on Machine Learning*, 2014, pp. 1899–1907.

[13] T. Luong, R. Socher, and C. D. Manning, "Better word representations with recursive neural networks for morphology," in *Proceedings of the Seventeenth Conference on Computational Natural Language Learning*, 2013, pp. 104–113.

[14] R. Soricut and F. Och, "Unsupervised morphology induction using word embeddings," in *Proceedings of the 2015 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies*, 2015, pp. 1627–1637.

[15] D. Lawrie, H. Feild, and D. Binkley, "Extracting meaning from abbreviated identifiers," in *Proceedings of IEEE International Working Conference on Source Code Analysis and Manipulation*. IEEE, 2007, pp. 213–222.

[16] D. Lawrie, D. Binkley, and C. Morrell, "Normalizing source code vocabulary," in *Proceedings of 2010 17th Working Conference on Reverse Engineering*. IEEE, 2010, pp. 3–12.

[17] B. Dit, L. Guerrouj, D. Poshyvanyk, and G. Antoniol, "Can better identifier splitting techniques help feature location?" in *Proceedings of 2011 IEEE 19th International Conference on Program Comprehension*. IEEE, 2011, pp. 11–20.

[18] D. Lawrie and D. Binkley, "Expanding identifiers to normalize source code vocabulary," in *Proceedings of 2011 27th IEEE International Conference on Software Maintenance*. IEEE, 2011, pp. 113–122.

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

[20] Y. Tian, D. Lo, and J. Lawall, "Automated construction of a software-specific word similarity database," in *Proceedings of 2014 Software Evolution Week-IEEE Conference onSoftware Maintenance, Reengineering and Reverse Engineering*. IEEE, 2014, pp. 44–53.

[21] J. Yang and L. Tan, "Swordnet: Inferring semantically related words from software context," *Empirical Software Engineering*, vol. 19, no. 6, pp. 1856–1886, 2014.

[22] 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*. IEEE Press, 2013, pp. 377–386.

[23] G. A. Miller, "Wordnet: a lexical database for english," *Communications of the ACM*, vol. 38, no. 11, pp. 39–41, 1995.

[24] C. Chen, Z. Xing, and X. Wang, "Unsupervised software-specific morphological forms inference from informal discussions," in *Proceedings of the 39th International Conference on Software Engineering*. IEEE Press, 2017, pp. 450–461.

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

[26] P. Bojanowski, E. Grave, A. Joulin, and T. Mikolov, "Enriching word vectors with subword information," *arXiv preprint arXiv:1607.04606*, 2016.

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

[28] Y. Park, S. Patwardhan, K. Visweswariah, and S. C. Gates, "An empirical analysis of word error rate and keyword error rate." in *Proceedings of Annual Conference of the International Speech Communication Association*, 2008, pp. 2070–2073.

[29] S. P. Ponzetto and M. Strube, "Knowledge derived from wikipedia for computing semantic relatedness." *Journal of Artificial Intelligence Research*, vol. 30, pp. 181–212, 2007.

[30] ——, "Deriving a large scale taxonomy from wikipedia," in *Proceedings of AAAI Conference on Artificial Intelligence*, 2007, pp. 1440–1445.

[31] 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 *Proceedings of 2016 IEEE 23rd International Conference on Software Analysis, Evolution, and Reengineering*. IEEE, 2016, pp. 90–101.

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

[33] Y. Huang, C. Chen, Z. Xing, T. Lin, and Y. Liu, "Tell them apart: distilling technology differences from crowd-scale comparison discussions," in *Proceedings of the 33rd ACM/IEEE international conference on automated software engineering. ACM*, 2018, pp. 214–224.

[34] X. Wang, C. Chen, and Z. Xing, "Domain-specific machine translation with recurrent neural network for software localization," *Empirical Software Engineering*, pp. 1–32, 2019.

[35] S. Gao, C. Chen, Z. Xing, Y. Ma, W. Song, and S.-W. Lin, "A neural model for method name generation from functional description," in *2019 IEEE 26th International Conference on Software Analysis, Evolution and Reengineering (SANER)*. IEEE, 2019, pp. 414–421.

[36] V. I. Levenshtein, "Binary codes capable of correcting deletions, insertions, and reversals," *Soviet physics doklady*, vol. 10, no. 8, pp. 707–710, 1966.

[37] F. J. Damerau, "A technique for computer detection and correction of spelling errors," *Communications of the ACM*, vol. 7, no. 3, pp. 171–176, 1964.

[38] C. Chen, S. Gao, and Z. Xing, "Mining analogical libraries in q&a discussions–incorporating relational and categorical knowledge into word embedding," in *Proceedings of 2016 IEEE 23rd International Conference on Software Analysis, Evolution, and Reengineering,*. IEEE, 2016, pp. 338–348.

[39] R. Mandala, T. Takenobu, and T. Hozumi, "The use of wordnet in information retrieval," in *Proceedings of the Conference Use of WordNet in Natural Language Processing Systems:*, 1998, pp. 31–37.

[40] A. Hliaoutakis, G. Varelas, E. Voutsakis, E. G. Petrakis, and E. Milios, "Information retrieval by semantic similarity," *International journal on semantic Web and information systems*, vol. 2, no. 3, pp. 55–73, 2006.

[41] C. Chen and Z. Xing, "Towards correlating search on google and asking on stack overflow," in *2016 IEEE 40th Annual Computer Software and Applications Conference (COMPSAC)*, vol. 1. IEEE, 2016, pp. 83–92.

[42] C. Chen, Z. Xing, and Y. Liu, "By the community & for the community: a deep learning approach to assist collaborative editing in q&a sites," *Proceedings of the ACM on Human-Computer Interaction*, vol. 1, no. CSCW, p. 32, 2017.

[43] C. Chen, X. Chen, J. Sun, Z. Xing, and G. Li, "Data-driven proactive policy assurance of post quality in community q&a sites," *Proceedings of the ACM on Human-Computer Interaction*, vol. 2, no. CSCW, p. 33, 2018.

[44] R. Singh and N. S. Mangat, *Elements of survey sampling*. Springer Science & Business Media, 2013, vol. 15.

[45] J. Cohen, "A coefficient of agreement for nominal scales," *Educational and Psychological Measurement*, vol. 20, no. 1, pp. 37–46, 1960. [Online]. Available: https://doi.org/10.1177/001316446002000104

[46] J. R. Landis and G. G. Koch, "The measurement of observer agreement for categorical data," *biometrics*, pp. 159–174, 1977.

[47] C. D. Manning, P. Raghavan, H. Schütze *et al.*, *Introduction to information retrieval*. Cambridge university press Cambridge, 2008, vol. 1.

[48] M. Gegick, P. Rotella, and T. Xie, "Identifying security bug reports via text mining: An industrial case study," in *Proceedings of 2010 7th IEEE working conference on Mining software repositories*. IEEE, 2010, pp. 11–20.

[49] 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*. IEEE Press, 2015, pp. 869–879.

[50] E. Linstead, P. Rigor, S. Bajracharya, C. Lopes, and P. Baldi, "Mining concepts from code with probabilistic topic models," in *Proceedings of the IEEE/ACM international conference on Automated software engineering*. ACM, 2007, pp. 461–464.

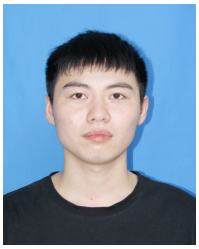
[51] M. F. Porter, "An algorithm for suffix stripping," *Program*, vol. 14, no. 3, pp. 130–137, 1980.

[52] L. Mamykina, B. Manoim, M. Mittal, G. Hripcsak, and B. Hartmann, "Design lessons from the fastest q&a site in the west," in *Proceedings of the SIGCHI conference on Human factors in computing systems*. ACM, 2011, pp. 2857–2866.

[53] 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*. Springer, 2009, pp. 94–125.
[54] A. C. Finkelstein, D. Gabbay, A. Hunter, J. Kramer, and B. Nuseibeh, "Inconsistency handling in multiperspective specifications," *IEEE Transactions on Software Engineering*, vol. 20, no. 8, pp. 569–578, 1994.
[55] A. Corazza, S. Di Martino, and V. Maggio, "Linsen: An efficient approach to split identifiers and expand abbreviations," in *Proceedings of the 28th IEEE International Conference on Software Maintenance*. IEEE, 2012, pp. 233–242.
[56] 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*. ACM, 2008, p. 7.
[57] E. Enslen, E. Hill, L. Pollock, and K. Vijay-Shanker, "Mining source code to automatically split identifiers for software analysis," in *Proceedings of 2009 6th IEEE International Working Conference on Mining Software Repositories*. IEEE, 2009, pp. 71–80.
[58] H. Liu, Z.-Z. Hu, J. Zhang, and C. Wu, "Biothesaurus: a web-based thesaurus of protein and gene names," *Bioinformatics*, vol. 22, no. 1, pp. 103–105, 2006.
[59] 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*. ACM, 2007, pp. 212–224.
[60] G. Sridhara, E. Hill, L. Pollock, and K. Vijay-Shanker, "Identifying word relations in software: A comparative study of semantic similarity tools," in *Proceedings of The 16th IEEE International Conference on Program Comprehension*. IEEE, 2008, pp. 123–132.
[61] 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*. ACM, 2006, pp. 3–14.
[62] S. Wang, D. Lo, and L. Jiang, "Inferring semantically related software terms and their taxonomy by leveraging collaborative tagging," in *Proceedings of 2012 28th IEEE International Conference on Software Maintenance*. IEEE, 2012, pp. 604–607.
[63] G. Chen, C. Chen, Z. Xing, and B. Xu, "Learning a dual-language vector space for domain-specific cross-lingual question retrieval," in *Proceedings of 31st IEEE/ACM International Conference on Automated Software Engineering*. IEEE, 2016, pp. 744–755.
[64] C. Chen, Z. Xing, Y. Liu, and K. L. X. Ong, "Mining likely analogical apis across third-party libraries via large-scale unsupervised api semantics embedding," *IEEE Transactions on Software Engineering*, 2019.
[65] 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*. ACM, 2016, pp. 834–839.
[66] C. Chen, Z. Xing, and Y. Liu, "Whats spains paris? mining analogical libraries from q&a discussions," *Empirical Software Engineering*, pp. 1–40, 2018.
[67] S. Beyer and M. Pinzger, "Synonym suggestion for tags on stack overflow," in *Proceedings of the 2015 IEEE 23rd International Conference on Program Comprehension*. IEEE Press, 2015, pp. 94–103.
[68] ——, "Grouping android tag synonyms on stack overflow," in *Proceedings of the 13th International Workshop on Mining Software Repositories*. ACM, 2016, pp. 430–440.

Xiang Chen received the B.Sc. degree in the school of management from Xi'an Jiaotong University, China in 2002. Then he received the M.Sc., and Ph.D. degrees in computer software and theory from Nanjing University, China in 2008 and 2011 respectively. He is with the School of Information Science and Technology at Nantong University as an associate professor. His research interests are mainly in software maintenance and software testing, such as mining software repository, software defect prediction, combinatorial testing, regression testing, and spectrum based fault localization. He has published over 40 papers in referred journals or conferences, including Information and Software Technology, Journal of Systems and Software, Journal of Software: Evolution and Process, IEEE Transactions on Reliability, Software Quality Journal, COMPSAC and APSEC. He is a senior member of CCF, China and a member of IEEE and ACM. https://smartse.github.io/index.html

Chunyang Chen is a lecturer (Assistant Professor) in Faculty of Information Technology, Monash University, Australia. His research focuses on software engineering, deep learning and human-computer interaction. He has published over 16 papers in referred journals or conferences, including Empirical Software Engineering, ICSE, ASE, CSCW, ICSME, SANER. He is a member of IEEE and ACM. He received ACM SIGSOFT distinguished paper award in ASE 2018, best paper award in SANER 2016, and best tool demo in ASE 2016. https://chunyang-chen.github.io/



Dun Zhang received the B.S. degree in computer science from Nantong University, Nantong, in 2017. He is currently pursuing the Master degree with School of Information Science and Technology, Nantong University. His research interest is mining software repository.

Zhenchang Xing is the senior lecturer at the research school of computer science, Australian National University, Australia. Dr. Xings research interests include software engineering and human-computer interaction. His work combines software analytic, behavioral research methods, data mining techniques, and interaction design to understand how developers work, and then build recommendation or exploratory search systems for the timely or serendipitous discovery of the needed information.