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.2018.2876006, IEEE Transactions on Software Engineering

Bridging Semantic Gaps between Natural Languages and APIs with Word Embedding

Xiaochen Li, He Jiang, *Member, IEEE,* Yasutaka Kamei, *Member, IEEE,* and Xin Chen,

Abstract—Developers increasingly rely on text matching tools to analyze the relation between natural language words and APIs. However, semantic gaps, namely textual mismatches between words and APIs, negatively affect these tools. Previous studies have transformed words or APIs into low-dimensional vectors for matching; however, inaccurate results were obtained due to the failure of modeling words and APIs simultaneously. To resolve this problem, two main challenges are to be addressed: the acquisition of massive words and APIs for mining and the alignment of words and APIs for modeling. Therefore, this study proposes Word2API to effectively estimate relatedness of words and APIs. Word2API collects millions of commonly used words and APIs from code repositories to address the acquisition challenge. Then, a shuffling strategy is used to transform related words and APIs into tuples to address the alignment challenge. Using these tuples, Word2API models words and APIs simultaneously. Word2API outperforms baselines by 10%-49.6% of relatedness estimation in terms of precision and NDCG. Word2API is also effective on solving typical software tasks, e.g., query expansion and API documents linking. A simple system with Word2API-expanded queries recommends up to 21.4% more related APIs for developers. Meanwhile, Word2API improves comparison algorithms by 7.9%-17.4% in linking questions in Question&Answer communities to API documents.

Index Terms—Relatedness Estimation, Word Embedding, Word2Vec, Query Expansion, API Documents Linking

---

1 Introduction

SOFTWARE developers put considerable efforts to study APIs (Application Programming Interfaces) [1], [2]. To facilitate this process, many tools have been developed to retrieve information about APIs, e.g., searching API sequences based on a query [3] or recommending API documents for answering technical questions [4]. These tools generally utilize information retrieval models, such as Vector Space Model (VSM) [4], [5], [6], to transform queries and APIs into words and conduct text matching to find required APIs or API documents [7]. Since there is usually a mismatch between the content of natural languages and APIs, the performance of these tools is negatively affected [7].

For example, in the task of API sequences recommendation, when a developer searches for APIs implementing 'generate md5 hash code', Java APIs of 'MessageDigest#getInstance' and 'MessageDigest#digest' may be required [8]. However, neither the word 'md5' nor 'hash code' could be matched with these APIs, which misleads information retrieval models to return the required APIs [7].

Another example is from the task of API documents linking. Developers usually ask technical questions on Question & Answer communities, e.g., 'How to (conduct a) sanity check (on) a date in Java'.[1] In their answers, the API 'Calendar#setLenient' is recommended by participants. However, based on text matching, the relationship between 'sanity check (on) a date' and 'Calendar#setLenient' is difficult to be determined. The question submitter even complained that '(it is) not so obvious to use lenient calendar'.

In the above examples, the mismatches between natural language words and APIs are semantic gaps. The gaps hinder developers from using APIs [9] and tend to bring thousands of defects in API documents [10]. They are also a major obstacle for the effectiveness of many software engineering tools [7], [11]. Previous studies have shown that a text-matching based retrieval tool could only return 25.7% to 38.4% useful code snippets in top-10 results for developers' queries [7]. To bridge the gaps, a fundamental solution is to correctly estimate the relatedness or similarity between a word and an API or a set of words and APIs, e.g., generating accurate similarity between words 'sanity check (on) a date' and the API 'Calendar#setLenient'.

Motivated by the aim of achieving such a solution, many algorithms for relatedness estimation have been proposed, including latent semantic analysis [12], co-occurrence analysis [11], WordNet thesaurus [13], etc. Among them, word embedding has recently shown its advantages [4], [14]; it constructs low-dimensional vectors of words or APIs for relatedness estimation. Existing studies tried to train software word embedding [4] based on Java/Eclipse tutorials and user guides, as well as API embedding [14] with API sequences from different programming languages. These strategies may still be ineffective to estimate the words-APIs relatedness, as they only learn the relationships for either words or APIs.

To improve the performance of existing solutions, it is necessary to model the words and APIs simultaneously into the same vector space. However, two main challenges are to

- X. Li and H. Jiang are with School of Software, Dalian University of Technology, Dalian, China, and Key Laboratory for Ubiquitous Network and Service Software of Liaoning Province. H. Jiang is also an adjunct professor in Beijing Institute of Technology. E-mail: li1989@mail.dlut.edu.cn, jianghe@dlut.edu.cn (corresponding email)
- Y. Kamei is with the Principles of Software Languages Group (POSL), Kyushu University, Japan. Email: kamei@ait.kyushu-u.ac.jp
- X. Chen is with School of Computer Science and Technology, Hangzhou Dianzi University. E-mail: chenxin4391@mail.dlut.edu.cn

1. https://stackoverflow.com/questions/226910/

0098-5589 (c) 2018 IEEE. Personal use is permitted, but republication/redistribution requires IEEE permission. See http://www.ieee.org/publications_standards/publications/rights/index.html for more information.

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.2018.2876006, IEEE Transactions on Software Engineering

be addressed: the acquisition challenge and the alignment challenge. The acquisition challenge is how to collect a large number of documents that contain diverse words and APIs. API tutorials and user guides are usually full of words, but have few APIs. The alignment challenge is how to align words and APIs to fully mine their overall relationship in a fixed window size, since word embedding mines word-API relationships based on the co-occurrence of words and APIs.

In this study we propose Word2API to address the two challenges. Word2API first collects large-scale files with source code and method comments from GitHub[2] to address the acquisition challenge. Source code and method comments usually contain diverse words and APIs commonly used by developers. Then, Word2API preprocesses these files. It extracts words in method comments and APIs in source code with a set of heuristic rules, which are efficient in identifying semantically related words and APIs in the files. After that, the extracted words and APIs regarding the same method are combined as a word-API tuple. Since the method comment always comes before the API calls in a method[3], the co-occurrence of words and APIs may be hardly mined in a fixed window. Word2API leverages a shuffling strategy to address the alignment challenge. This strategy randomly shuffles words and APIs in a word-API tuple to form a shuffled tuple for training. Since there is valuable information among all words and APIs in the same word-API tuple, this strategy is effective in increasing the word-API collocations and revealing the overall relationship between words and APIs in a fixed window. Finally, Word2API applies word embedding on the shuffled results to generate word and API vectors.

We trained Word2API with 391 thousand Java projects consisting of more than 31 million source code files from GitHub. Word2API generates vectors for 89,422 words and 37,431 APIs. We evaluate Word2API by recommending semantically related APIs for a word. For 31 out of 50 words, the top-1 recommended API is related, which outperforms comparison algorithms by 10%-49.6% in terms of precision and Normalized Discounted Cumulative Gain (NDCG). Meanwhile, the shuffling strategy significantly improves the effectiveness of word embedding in constructing semantically related vectors from word-API tuples.

Besides, we demonstrate two applications of Word2API, including query expansion for API sequences recommendation [3] and API documents linking [4]. API sequences recommendation recommends API sequences in source code for a user query. API documents linking links questions in Q&A communities to the API documents that may be useful to answer the questions. For the first task, Word2API expands a user query into a set of APIs. A simple system with Word2API-expanded queries can recommend up to 21.4% more related API sequences than baseline algorithms. For the second task, Word2API outperforms existing algorithms by 8.9% and 7.9% in linking useful API documents to questions in Stack Overflow in terms of Mean Average Precision (MAP) and Mean Reciprocal Rank (MRR) respectively.

To conclude, we make the following contributions.

1) We propose Word2API to solve the problem of constructing low-dimensional representations for both words and APIs simultaneously. Word2API successfully addresses the acquisition challenge and alignment challenge in this problem.
2) With Word2API, we generate 126,853 word and API vectors to bridge the sematic gaps between natural language words and APIs. We publish the generated vectors as a dictionary for research.[4]
3) We show two applications of Word2API. Word2API improves the performance of two typical software engineering tasks, i.e., API sequences recommendation and API documents linking.

Outline. Section 2 presents the background of this study. Section 3 shows the framework of Word2API. Experimental settings and results on relatedness estimation are introduced in Sections 4 and 5 respectively. Two applications of Word2API are shown in Sections 6 and 7. In Section 8, threats to validity are discussed. We review the related work in Section 9. Finally, Section 10 concludes this paper.

2 Background

2.1 Terminology

This subsection defines the major terms used in this paper.

APIs are pre-defined functions for communication between software components [15]. They are designed under the criteria of high readability, reusability, extendibility, etc. [16]. In this study, an API refers to a method-level API that consists of the fully qualified name of an API type and a method name.

A *word* is a natural language element in a document or text to express human intentions. We take all the non-API elements in a document or text as words. In software engineering, there are many API-like words [4] such as 'readLine', 'IOException', etc. We also call them words.

In addition, '*term*' is used to generally indicate either *APIs* or natural language *words*.

We use the word '*document*' to indicate a text with many words or APIs. Some special documents in software engineering are *API documents* [4]. In this study, API documents refer to the documents in API specifications. Each API document contains method-level APIs in the same class and illustrates the class-description, method-description, etc.

2.2 Word Embedding

Word embedding is a fundamental component of Word2API. It was originally designed to transform words in word sequences into low-dimensional vectors [17]. Many models have been proposed to implement word embedding, e.g., Continuous Bag-of-Words model (CBOW) [18], continuous Skip-gram model (Skip-gram) [17], etc. To facilitate the use of these models, Google publishes a tool[5] that implements the CBOW and Skip-gram models. We take the CBOW model as an example to explain word embedding, as it is the default model in the word embedding tool.

2. GitHub. https://github.com/
3. In this paper, 'method' refers to a function or procedure defined in a class. We use 'algorithm' or 'approach' to describe Word2API
4. The dictionary $W2A_{DIC}$. https://github.com/softw-lab/word2api
5. Google tool. https://code.google.com/archive/p/word2vec/

0098-5589 (c) 2018 IEEE. Personal use is permitted, but republication/redistribution requires IEEE permission. See http://www.ieee.org/publications_standards/publications/rights/index.html for more information.

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.2018.2876006, IEEE Transactions on Software Engineering

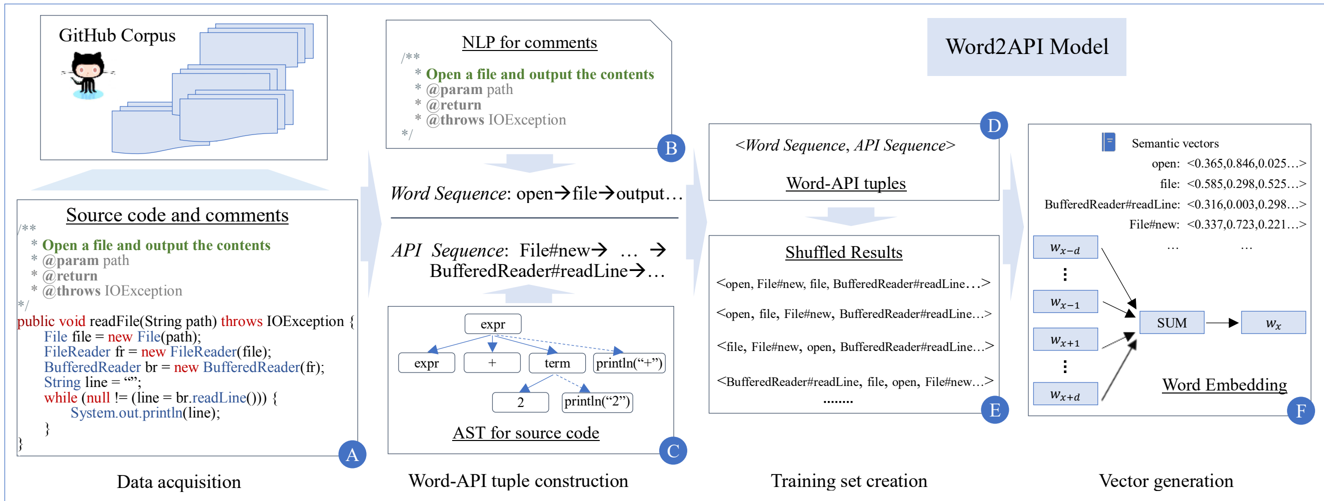


Fig. 2: Framework of Word2API model. (A) crawls the source code and comments from GitHub. (B) and (C) extract the word sequences and API sequences in the crawled data. (D) combines the sequences as a tuple. (E) shuffles the tuples to generate an unlabeled training set. (F) applies word embedding on the training set to get the term vectors.

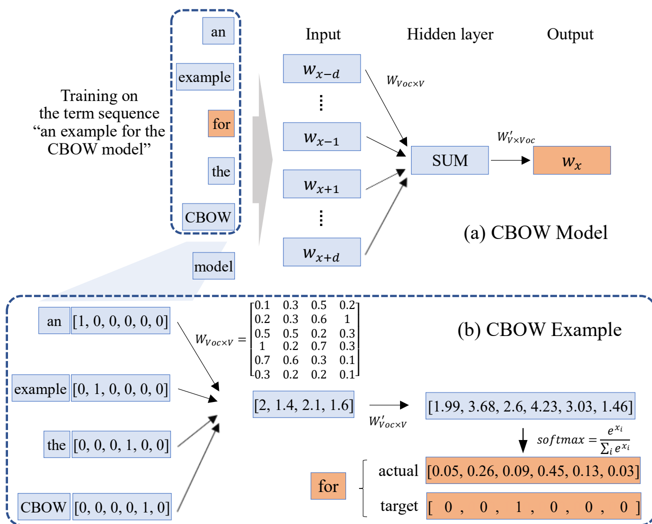


Fig. 1: CBOW model for word embedding.

CBOW is a neural network model to learn word representations from an unlabeled training set [18]. Fig. 1(a) presents the framework of CBOW. CBOW consists of an input layer, an output layer, and a hidden layer. The hidden layer h is a $1 \times V$ vector to represent words in a low-dimensional space. V is pre-defined by users. CBOW uses a matrix $W_{Voc \times V}$ to propagate information between layers, where Voc is the vocabulary of the training set.

Initially, we randomly initialize the values of $W_{Voc \times V}$ and represent each word x in Voc with a one-hot vector w_x. The one-hot vector is a zero vector with the exception of a single 1 to uniquely identify the word (Fig. 1(b)). The vector length is the same as the vocabulary size $|Voc|$.

With these one-hot vectors, CBOW tries to predict the center word with its surrounding context in a fixed window size d. Specifically, CBOW takes in the vectors of the surrounding words $W_x^d = \{w_{x-d}, \ldots, w_{x-1}, w_{x+1}, \ldots, w_{x+d}\}$ in a $2d$ sized window as the input and the vector of the center word w_x as the target output. For example, if $d = 2$, $V = 4$ and 'for' is the center word, then the input includes the vectors of 'an', 'example', 'the', 'CBOW'. Based on $W_{Voc \times V}$, CBOW propagates the input to the hidden layer h

$$h = \frac{1}{2d}(w_{x-d} + \ldots + w_{x-1} + w_{x+1} + \ldots + w_{x+d}) \cdot W_{Voc \times V} \quad (1)$$

Then, the vector in h continues forward propagating according to the parameter matrices $W'_{Voc \times V}$:

$$w_{1 \times Voc} = softmax(h \cdot W'_{Voc \times V}), \quad (2)$$

where $w_{1 \times Voc}$ is the actual output of the center word. For example, the network outputs a vector [0.05, 0.26, 0.09, 0.45, 0.13, 0.03] in Fig. 1(b). Since $w_{1 \times Voc}$ is far different from the target output w_x=[0, 0, 1, 0, 0, 0], CBOW aims to maximize the average probability that the actual output is w_x:

$$L_M = \frac{1}{X} \sum_{x=1}^{X} \log p(w_x | W_x^d) \quad (3)$$

CBOW optimizes the output by tuning the parameter matrix $W_{Voc \times V}$ with back propagation. After training, we get the values of the final parameter matrix. For a word x, the low-dimensional vector is calculated as $w_x \cdot W_{Voc \times V}$.

3 The Word2API Model

Word2API represents natural language words and APIs with low-dimensional vectors. As depicted in Fig. 2, Word2API consists of four steps, including data acquisition, word-API tuple construction, training set creation, and vector generation. We detail these steps in this section.

3.1 Data Acquisition

To train the vectors for words and APIs, we construct a large-scale corpus with source code and method comments. The corpus (referred as GitHub corpus) is constructed from the Java projects created from 2008 to 2016 on GitHub. We analyze Java projects as they have a broad impact on

0098-5589 (c) 2018 IEEE. Personal use is permitted, but republication/redistribution requires IEEE permission. See http://www.ieee.org/publications_standards/publications/rights/index.html for more information.

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.2018.2876006, IEEE Transactions on Software Engineering

software development. However, Word2API is independent of programming languages. We download the zip snapshots in July 2017 of these projects with GitHub APIs.[6] We exclude the projects with zero stars, since they are usually toy or experimental projects [8]. For each project, all the Java files are extracted. Each file consists of several methods and their comments (Fig. 2(A)). In total, we collect 391,690 Java projects with 31,211,030 source code files. It should be noted that, in GitHub, a project may have many forks or third-party source code [19], leading to duplicate code snippets. We keep these duplications in the data set, as forking projects is a basic characteristic of GitHub.

3.2 Word-API Tuple Construction

With the GitHub corpus, we construct word-API tuples. A word-API tuple is a combination of a set of words and the corresponding APIs. We construct the tuples by analyzing the source code of these Java projects.

Specifically, we construct an AST (Abstract Syntax Tree) for each method in the source code by Eclipse's JDT Core Component.[7] In the AST, we extract the method comment (Fig. 2(B)) and its corresponding API types and method calls in the method body (Fig. 2(C)) to construct a word-API tuple. The word-API tuple consists of a word sequence extracted from the method comment and an API sequence obtained from API types and method calls in the method body.

For the method comment, we remove the HTML tags that match the regular expression '<.*?>', and split the sentences in the method comment by '. '. In Java language, sentences in a method comment are typically enclosed between '/**' and '*/' above the method body. We extract the words in the first sentence to make up the word sequence portion of a word-API tuple, since this sentence is usually a semantically related high-level summary of a method [8].

For the method body, we extract Java Standard Edition (SE) API types and method calls to make up the API sequence portion of the word-API tuple. We note that a method is usually implemented with many syntactic units [14], including APIs, variables/identifiers, literals, etc. Java SE APIs may not fully reveal the intents of a method comment. However, they are still semantically related to the comment [8]. We extract Java SE APIs as follows:

- We traverse the AST of a method to collect the APIs for class instance creation and method calls. We represent these APIs with their fully qualified names by resolving the method binding. If an API is the argument of another API, we represent the API in the argument list first. For example, 'BufferedReader br = new BufferedReader(new FileReader()); br.readLine())' is represented as 'java.io.FileReader#new, java.io.BufferedReader#new, and java.io.BufferedReader#readLine'. We omit the return type and argument types in this representation, since the overloaded APIs of different return types or argument types usually convey the same semantic meaning [20].
- We extract Java SE APIs from the collected APIs by matching their package names with the ones in the Java SE API specification[8] (also called API references). We delete the tuples without Java SE APIs.

After the above process, a set of word-API tuples are achieved. We assume that the word sequence in each tuple summarizes the behaviors or purposes of the corresponding APIs. However, besides summarizing APIs, developers may also add TODO lists, notes, etc. in the method comments [21], which are noises in our scenario. Therefore, we filter out these tuples, if the word sequence in a tuple:

- starts with 'TODO', 'FIXME', 'HACK', 'REVISIT', 'DOCUMENTME', 'XXX'; these tags are commonly used for task annotations instead of summarizing APIs [22], e.g., 'TODO remove this';
- starts with words like 'note', 'test'; developers use these words to write an explanatory or auxiliary comments [8], [23], e.g., 'testing purpose only';
- is a single word instead of a meaningful sentence.

For the remaining word sequences, we perform tokenization [24], stop words removal[9] and stemming [25]. We remove words that are numbers or single letters. If a word is an API-like word, we split it according to its camel style, e.g., splitting 'nextInt' into 'next' and 'int'. Finally, 13,883,230 tuples are constructed (Fig. 2(D)).

3.3 Training Set Creation

This step creates an unlabeled training set with the constructed word-API tuples for word embedding. Word embedding is a co-occurrence based method that analyzes the relationship of terms in a fixed window size. Word embedding works well in a monolingual scenario, e.g., sequential natural language words [4], source code identifiers [26], and API sequences [14], since words or APIs nearby have strong semantic relatedness. In contrast, it may be hard for word embedding to capture the co-occurrences between words and APIs in a bilingual scenario such as comments and their corresponding APIs. In this scenario, words and APIs usually do not appear within each other's window, e.g., words in the method comments always come before the APIs. The problem mainly comes from the word-API tuples we collected. However, to the best of our knowledge, no training set could be directly used for effectively mining word embedding for both words and APIs like in a monolingual scenario. An ideal training set should both have a large number of words and APIs and properly align semantic relatedness collocations of words and APIs. Since word-API tuples consist of diverse words and APIs frequently used by developers, the remaining challenge is, how to align words and APIs into a fixed window for relationship mining.

To resolve this problem, we merge words and APIs in the same tuple together and randomly shuffle them to create the training set. The shuffling step is to break the fixed location of words and APIs. It tries to obtain enough collocations between each word/API and other APIs/words. To increase semantically related collocations, we repeat the

6. GitHub APIs. https://developer.github.com/v3/
7. Eclipse JDT Core Component. http://www.eclipse.org/jdt/core/
8. Java SE API Spec. http://docs.oracle.com/javase/8/docs/api/
9. Default English stop words. http://www.ranks.nl/stopwords

0098-5589 (c) 2018 IEEE. Personal use is permitted, but republication/redistribution requires IEEE permission. See http://www.ieee.org/publications_standards/publications/rights/index.html for more information.

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.2018.2876006, IEEE Transactions on Software Engineering

shuffling step ten times to generate ten shuffled copies of an original word-API tuple. Fig. 2(E) is the shuffled results of the word-API tuple created from Fig. 2(B) and Fig. 2(C). After shuffling, words and APIs tend to co-occur in a small window. We take these shuffled results as the training set for word embedding. The training set contains 138,832,300 shuffled results. Its size is more than 30 gigabyte.

The implementation of the shuffling step can be understood from two perspectives. From a training set perspective, this step transforms the original word-API tuples into shuffled tuples and uses a classical CBOW model to learn word embedding. From a model perspective, the shuffling step is equivalent to a modified CBOW model, where the surrounding words for recovering a center word are not selected based on the window but are randomly sampled from the entire word-API tuple.

The underlying reason of the above procedure is that words and APIs in the same word-API tuple tend to contain valuable semantic information (relatedness) for mining. The shuffling strategy increases the information interaction and helps word embedding learn the knowledge of collocations between words and APIs in a tuple. After shuffling, the collocations of words and APIs increase, i.e., words and APIs have higher chances to appear within each other's window. Hence, word embedding could learn the overall knowledge of each tuple. Since the shuffling is random, we repeat the shuffling step to increase related word-API collocations. We evaluate the shuffling step in Section 5.3.

3.4 Vector Generation

The last step of Word2API is to train a word embedding model with the training set for vector generation. We utilize the word embedding tool for unsupervised training. Word embedding models have many parameters, e.g., 'window size', 'vector dimension', etc. Although previous studies show that task-specific parameter optimization influences algorithm performance [27], such optimization may threaten the generalization of an algorithm. Hence, in this study, all the parameters in the tool are set to the default ones except the '-min-count' (the threshold to discard a word or API). Since we generate ten shuffled results for a tuple, the parameter '-min-count' is set to 50 instead of the default value of 5. It means that we discard all the words and APIs that appear less than 50 times in the training set. For some important parameters, we train word embedding with the default model CBOW, a more efficient model compared to the Skip-gram model in the word embedding tool[10]. The default window size is 5 and the dimension of the generated vectors is 100. The window size determines how many words or APIs nearby are considered as co-occurred and the vector dimension reflects the dimension of the generated vector for each word or API. The other parameters are listed as follows:

- 'sample' is 1e-3: the threshold to down-sample a high-frequency term. The word embedding tool down-samples a term t_i in the training set by $P(t_i) = (\sqrt{z(t_i)/sample}+1) \times sample/z(t_i)$[11], where $z(t_i)$ is the probability that term i appears in the training set and $P(t_i)$ is the probability to keep this term in the training set. When a term appears frequently, $P(t_i)$ tends to be small, which means the probability to keep this term in the training set is low.

- 'hs' is 0: hierarchical softmax is not used for training.

- 'negative' is 5: the number of random-selected negative samples in a window.

- 'iter' is 5: the number of times to iterate the training set.

- 'alpha' is 0.05: the starting learning rate.

- 'thread' is 32: the number of threads for training.

After running the word embedding tool, 89,422 word vectors and 37,431 API vectors are generated eventually. These vectors are important to bridge the semantic gaps between natural language words and APIs. For this purpose, we define word-API similarity and words-APIs similarity:

Word-API Similarity is the similarity between a word w and an API a. It is the cosine similarity of vectors $\overrightarrow{V_w}$ and $\overrightarrow{V_a}$:

$$sim(w,a) = \frac{\overrightarrow{V_w} \cdot \overrightarrow{V_a}}{|\overrightarrow{V_w}||\overrightarrow{V_a}|}. \tag{4}$$

Words-APIs Similarity extends *Word-API Similarity* to a set of words W and a set of APIs A [28]:

$$sim(W,A)=\frac{1}{2}\left(\frac{\sum(sim_{max}(w,A)\times idf(w))}{\sum idf(w)}+\frac{\sum(sim_{max}(a,W)\times idf(a))}{\sum idf(a)}\right), \tag{5}$$

where $sim_{max}(w, A)$ returns the highest similarity between w and each API $a \in A$, and $idf(w)$ is calculated as the number of documents (word sequences in word-API tuples) divided by the number of documents that contain w. Similarly, $sim_{max}(a, W)$ and $idf(a)$ can be defined.

4 Evaluation Setting

In this section, we detail the settings for evaluating Word2API, including Research Questions (RQs), baseline algorithms, the evaluation strategy, and evaluation metrics.

4.1 Research Questions

RQ1: How does Word2API perform against the baselines in relatedness estimation between a word and an API?

To estimate term relatedness, many algorithms have been proposed. We compare Word2API with these algorithms to show the effectiveness of Word2API.

RQ2: How does Word2API perform under different settings?

For generalization, Word2API utilizes the default settings of the word embedding tool for vector generation. This RQ evaluates Word2API under different parameter settings.

RQ3: Does the shuffling step in training set creation contribute to the performance of Word2API?

We investigate whether the shuffling strategy can better train word and API vectors.

4.2 Baseline Algorithms for relatedness estimation

This part explains the main algorithms for relatedness estimation [11] and shows the baselines in this study.

10. We compare CBOW and Skip-gram in Sec. S1 of the supplement.
11. https://github.com/dav/word2vec/blob/master/src/word2vec.c

0098-5589 (c) 2018 IEEE. Personal use is permitted, but republication/redistribution requires IEEE permission. See http://www.ieee.org/publications_standards/publications/rights/index.html for more information.

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.2018.2876006, IEEE Transactions on Software Engineering

4.2.1 Latent Semantic Analysis (LSA)

LSA (also called Latent Semantic Indexing) [12] first represents the documents in a corpus with an $m \times n$ matrix. In the matrix, each row denotes a term in the corpus, each column denotes a document, and the value of a cell is the term weight in a document. Then, LSA applies Singular Value Decomposition to transform and reduce the matrix into an $m \times n'$ matrix. Each row of the matrix is an n'-dimensional vector that can be used to estimate the relatedness of different terms.

In this study, the inputs of LSA are word-API tuples. We take each tuple as a document. The value of a cell in the matrix is the frequency of a term in the document. Due to the large number of tuples (> 10 million), we randomly sample 20% tuples for training to resolve the computational problems in calculating high-dimensional matrices. n' is set to 200, since it achieves acceptable results on relatedness estimation [11]. We implement LSA with Matlab.

4.2.2 Co-occurrence based Methods

Co-occurrence based methods assume that terms are semantically related if they tend to co-occur in the same document or a fixed window size of the document. In this experiment, a document means a word-API tuple. Word2API belongs to this category. Besides, we highlight several other representative algorithms, including Pointwise Mutual Information (PMI), Normalized Software Distance (NSD), and Hyperspace Analogue to Language (HAL).

PMI measures term relatedness by comparing the probability of co-occurrence of two terms and the probability of occurrence of each term [29]. Co-occurrence means two terms co-occur in the same document regardless of the position and occurrence means a term occurs in a document. PMI of a word w and an API a is defined as:

$$PMI(w,a) = \log \frac{p(w,a)}{p(w)p(a)} \approx \log \frac{f(w,a)}{(f(w)) \times (f(a))}, \quad (6)$$

where $p(w,a)$ is the probability that w and a co-occur in a word-API tuple. It can be estimated by $f(w,a)$, namely the number of tuples that contain both w and a divided by the total tuples' number. $p(w)$ or $p(a)$ is the probability that w or a occurs in a tuple respectively, which can be estimated by $f(w)$ or $f(a)$ similarly.

NSD [11] calculates the similarity between a word w and an API a with the following formula [30]:

$$NSD(w,a) = \frac{\max\{\log(f(w)), log(f(a))\} - \log(|f(w) \bigcap f(a)|)}{\log(N) - \min\{\log(f(w)), log(f(a))\}}, \quad (7)$$

where $f(w)$ and $f(a)$ are the same definitions as those in formula (6) and N is the number of tuples.

HAL [31] constructs a high dimensional $n * n$ matrix to represent the co-occurrences of all the n terms in the word-API tuples. Each cell (row$_i$, column$_j$) in the matrix is the weight between term$_i$ and term$_j$, which is formalized as the Positive PMI (PPMI) between the corresponding terms [32]:

$$PPMI = \begin{cases} PMI(term_i, term_j), & PMI(term_i, term_j) > 0 \\ 0 & otherwise \end{cases} \quad (8)$$

TABLE 1: Selected words for evaluation

#	Word	#	Word	#	Word	#	Word	#	Word
1	agent	11	delete	21	key	31	random	41	tail
2	average	12	display	22	length	32	remote	42	thread
3	begin	13	environment	23	mp3	33	request	43	timeout
4	buffer	14	file	24	next	34	reserve	44	transaction
5	capital	15	filter	25	node	35	scale	45	uuid
6	check	16	graphics	26	object	36	select	46	validity
7	classname	17	http	27	open	37	session	47	word
8	client	18	input	28	parse	38	startup	48	xml
9	current	19	interrupt	29	port	39	string	49	xpath
10	day	20	iter	30	post	40	system	50	year

4.2.3 Thesaurus-based Methods

This line of methods uses linguistic dictionaries, e.g., WordNet, for relatedness estimation. However, such methods may be ineffective in software engineering areas [11], [33], due to the lack or mistaken definition of software-specific terms in the dictionaries, e.g., program reserved identifiers and APIs. We do not take them as baselines.

4.3 Evaluation Strategy

As to our knowledge, no dataset is publicly available for word-API relatedness estimation, as most evaluations depend on human judgements [11], [34]. We follow the widely accepted methodology of TREC[12] for evaluation [35], [36], a popular Text REtrieval Conference of over 25 years' history.

Given a corpus, TREC selects a set of queries for different algorithms to retrieve texts, e.g., web pages or documents. The results are ranked in a descending order. The top-k (usually, k=100 [35]) results are submitted to TREC. TREC merges the results from different algorithms and asks volunteers to judge the relatedness of the query-result pairs subjectively in a binary manner (related or unrelated). Similar to TREC, we conduct the evaluation as follows.

4.3.1 Word selection

This step selects a set of words as queries. Initially, we randomly select 50 words from the GitHub corpus. Among these words, nouns and verbs are selected as queries as they are more descriptive [23]. The other words are removed. The removed words are replaced by other randomly selected nouns or verbs until the number of words reaches 50 (in Table 1). This number is comparable to TREC [36] and other experiments in software engineering [8], [11].

This experiment selects 50 words for evaluation. It is a direct way to evaluate the semantic relatedness between words and APIs as suggested by previous studies [11], [37]. Since human usually have some intuitive understandings to APIs, the evaluation helps us understand whether the results returned by each algorithm are in accordance with the human intuition.

4.3.2 API collection

We run Word2API and the baseline algorithms with the selected words. For each word, we collect and merge the top-100 recommended APIs for evaluation.

12. Text REtrieval Conference TREC. http://trec.nist.gov/

0098-5589 (c) 2018 IEEE. Personal use is permitted, but republication/redistribution requires IEEE permission. See http://www.ieee.org/publications_standards/publications/rights/index.html for more information.

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.2018.2876006, IEEE Transactions on Software Engineering

4.3.3 Human judgement

Theoretically, there are 25,000 word-API pairs for judgements (50 words×5 algorithms×100 recommendations). Since some APIs may be recommended by more than one algorithm, there are 19,298 judgements eventually. Due to the large number of word-API pairs, we follow TREC to randomly split them into three non-overlapping partitions and assign the partitions to three volunteers for evaluation (related or unrelated). Each volunteer evaluates about 6,433 word-API pairs. The volunteers are graduate students, who have 3-5 years' experience in Java. We take them as junior developers. Since Junior developers (less than 5 years' experience) inhabit over 50% of all developers according to a survey[13] of 49,521 developers in Stack Overflow, the evaluation may be representative to the view of many developers.

The definition of relatedness is open [11]. Volunteers could consider the linguistic definition of a word, the usage scenarios of an API, etc. We ask volunteers to record how they understand each word during evaluation, i.e., the definition that they evaluate the word-API pairs. The judgements take 2 weeks. On average, 86 APIs are considered to be related to a word.

To evaluate the validity of human judgements, we randomly select a statistically significant sample for re-evaluation based on the total number of 19,298 word-API pairs with a confidence level of 99% and a confidence interval of 5% [38], resulting in a sample of 644 word-API pairs. We send the sample to a new volunteer for judgements. The Cohen's Kappa coefficient [39] between the first and second round of judgements is 0.636, which means that volunteers substantially agree on the judgements.

4.4 Evaluation Metrics

Based on the human judgements, we evaluate each algorithm from two aspects, namely, given a word, how many related APIs can be correctly recommended and whether the related APIs are ranked higher than the unrelated ones. For these aspects, precision and NDCG are employed [1], [40].

$$Precision@k = \frac{\text{\# of relevant APIs to word}_i \text{ in top-}k}{k}, \quad (9)$$

$$NDCG@k = \frac{DCG@k}{IDCG@k} \quad (DCG@k = \sum_{i=1}^{k} \frac{r_i}{\log_2 i + 1}), \quad (10)$$

where $r_i = 1$ if the ith API is related to the given word, and $r_i = 0$ otherwise. IDCG is the ideal result of DCG, which all related APIs in a ranking list rank higher than the unrelated ones. For example, if an algorithm recommends five APIs in which the 2nd, 4th APIs are related, we can represent the results as {0,1,0,1,0}. Then the ideal result is {1,1,0,0,0}.

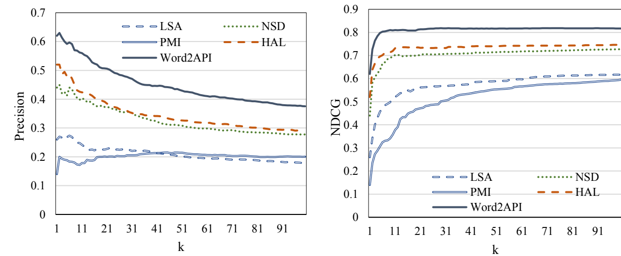
5 Evaluation Results

5.1 Answer to RQ1: Baseline Comparison

5.1.1 Precision and NDCG

Fig. 3(a) and Fig. 3(b) are the averaged precision and NDCG for different algorithms over the selected 50 words respectively. The x-axis is the ranking list size k from 1 to 100 and y-axis is precision and NDCG on varied k.

13. https://insights.stackoverflow.com/survey/2016



(a) Evaluation on precision (b) Evaluation on NDCG

Fig. 3: Precision and NDCG on 50 selected words.

In Fig. 3(a), Precision@1 of Word2API is 62%, which means that Word2API can find a semantically related API in the top-1 recommendation for 31 out of 50 query words. This result outperforms the best baseline algorithm by 10%. When recommending 20 APIs by Word2API, half of the APIs are semantically related. If we recommend 100 APIs, the precision of Word2API is still nearly 40%. Since there are about 86 related APIs for a selected word, the result means that Word2API finds nearly half of the related APIs. For NDCG, Word2API is superior to the other algorithms. NDCG@1, NDCG@2 and NDCG@6 of Word2API are 0.620, 0.726 and 0.803 respectively, which outperform the baselines by 0.102 to 0.496. We explore the statistical significance of the results with the paired Wilcoxon signed rank test over the entire ranking list, i.e., Precision@100 and NDCG@100.

H_0: *There is no significant difference between the performance of two algorithms over an evaluation metric.*

H_1: *There is significant difference between the performance of two algorithms over an evaluation metric.*

Since there are four baseline algorithms, the significance level is set to $0.05/4 = 1.25 \times 10^{-2}$ after Bonferroni correction [41]. The p-values on Precision@100 are 5.17×10^{-9}, 7.52×10^{-9}, 4.63×10^{-7}, 7.53×10^{-5} when comparing Word2API with LSA, PMI, NSD, HAL respectively. H_0 is rejected. Word2API significantly outperforms all the baseline algorithms. We also achieve the same conclusion for NDCG@100. The p-values are 1.77×10^{-8}, 2.99×10^{-9}, 8.76×10^{-6}, 5.20×10^{-4} for LSA, PMI, NSD, and HAL respectively.

For the baseline algorithms, HAL and NSD are the best, followed by LSA and PMI. Both HAL and NSD have been applied on software engineering tasks in previous studies [11], [32]. The two algorithms conduct relatedness estimation with high-dimensional vectors [32] or predefined functions [11]. The drawback of HAL and NSD is that they cannot refine the relatedness of two terms with other terms in the same context. In contrast, Word2API recovers a term based on the vectors of nearby terms. The recovering step is to mine and refine a term with the knowledge of its context. Hence, Word2API performs better over different metrics.

5.1.2 Examples of recommended APIs

Table 2 presents examples of the recommended APIs for words 'capital' and 'uuid'.[14] We omit the API package names for brevity. An API in bold is a related API by human judgements. Volunteers think the word 'capital' is

14. Other recommended APIs are at https://github.com/softw-lab/word2api

0098-5589 (c) 2018 IEEE. Personal use is permitted, but republication/redistribution requires IEEE permission. See http://www.ieee.org/publications_standards/publications/rights/index.html for more information.

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.2018.2876006, IEEE Transactions on Software Engineering

TABLE 2: Examples of top-10 recommended APIs for different algorithms.

	LSI	PMI	NSD	HAL	Word2API
capital	Character#getType	NSMException[1]#fillInStackTrace	**Character#toUpperCase**	**Character#toTitleCase**	**Character#toUpperCase**
	StringBuilder#insert	**Character#toTitleCase**	**Character#toTitleCase**	Character#isTitleCase	**Character#toTitleCase**
	Pattern#normalizeSlice	Character#offsetByCodePoints	**Character#isUpperCase**	String#getValue	**Character#isUpperCase**
	Pattern#normalizeClazz	**Character#toUpperCase**	Character#isLetter	**Character#isUpperCase**	**Character#toLowerCase**
	Character#toUpperCase	**Character#isUpperCase**	**Character#isLowerCase**	**Character#isTitleCaseImpl**	**Character#isLowerCase**
	StringBuilder#reverse	ToLongBiFunction<T>#applyAsLong	**Character#toLowerCase**	**Character#isLowerCase**	Character#isTitleCase
	StringBuilder#appendCP[2]	LongStream#sum	Character#offsetByCodePoints	CharacterData#toTitleCase	IndexedPropertyDescriptor#setReadMethod
	StringBuilder#setLength	**Vector<T>#subList**	StringBuilder#setCharAt	IAVException[3]#printStackTrace	StringBuilder#setCharAt
	StringBuilder#setCharAt	Character#isLetter	NSMException#fillInStackTrace	ITException[4]#fillInStackTrace	PropertyDescriptor#setName
	Character#isAlphabetic	**Character#isLowerCase**	String#codePointAt	**Character#toUpperCase**	**String#toUpperCase**
uuid	Objects#requireNonNull	**UUID#.new**	JAXBException#fillInStackTrace	**UUID#toString**	**UUID#randomUUID**
	Charset#newEncoder	**UUID#toString**	**UUID#padHex**	**UUID#.new**	**UUID#toString**
	CharSequence#equals	**UUID#randomUUID**	**UUID#md5**	**UUID#randomUUID**	**UUID#fromString**
	AssertionError#.new	**UUID#version**	**UUID#makeUuid**	**UUID#nameUUIDFromBytes**	**UUID#getLeastSignificantBits**
	UUID#toString	**UUID#getMostSignificantBits**	**UUID#generateUUIDString**	ThreadLocalRandom#nextBytes	**UUID#getMostSignificantBits**
	Supplier<T>#get	**UUID#getLeastSignificantBits**	**UUID#digits**	**Random#nextLong**	**UUID#digits**
	Scanner#hasNextShort	**UUID#fromString**	**UUID#version**	**UUID#equals**	**UUID#nameUUIDFromBytes**
	CharSequence#charAt	Long#intValue	**UUID#.new**	IIOMetadataNode#setNodeValue	SOAPEnvelope#createQName
	NullPointerException#.new	**UUID#nameUUIDFromBytes**	**UUID#timestamp**	Base64#getUrlEncoder	**UUID#makeUuid**
	TAccessor[5]#isSupported	**UUID#digits**	**UUID#nameUUIDFromBytes**	TemporalAdjusters#previousOrSame	**UUID#equals**

Note: [1]NSMException: NoSuchMethodException [2]appendCP: appendCodePoint [3]IAVException: InvalidAttributeValueException [4]ITException: InvocationTargetException [5]TAccessor: TemporalAccessor

semantically related with APIs that perform operations on the capital letters or first words. It is a concept that may be related to different API packages, e.g., 'String#toUpperCase' or 'Character#toUpperCase'. 'uuid' is considered to be related to APIs in the java.util.UUID package and some APIs for random number generation. It is a concept mainly related to a concrete package.

As shown in Table 2, the results of Word2API show similar understandings with volunteers. It associates 'capital' with APIs of 'Character#toUpperCase', 'Character#toLowerCase', and 'String#toUpperCase'. Although some related APIs are also detected by HAL, NSD and PMI, these algorithms still find some unrelated APIs in the top-5 results, e.g., 'String#getValue'. For the word 'uuid', many algorithms associate this concept with the UUID package. Word2API is among the best of these algorithms. In contrast, HAL fails to analyze this concept. Only half of the top-10 APIs are related to 'uuid'. The reason may be that HAL represents terms with high-dimensional vectors. The dimension equals to the vocabulary size. The high-dimensional representation increases the computation complexity which makes HAL unprecise [4], e.g., introducing noises and being dominated by dimensions with large entry values.

Conclusion. Word2API outperforms the baseline algorithms in capturing the word-API semantic relatedness.

5.2 Answer to RQ2: Parameter Influence

There are two main parameters for vector generation, namely the window size w and the vector dimension v.[15] This RQ generates variants of Word2API to evaluate the parameter influence. For the variants (in RQ2 and RQ3), additional human judgements are conducted on the new recommended APIs that have not been judged before.

15. We analyze the influence of the shuffling times, the number of iterations, the tuple length, etc. in Sec. S3 to S6 of the supplement.

5.2.1 Window Size

Fig. 4(a) shows precision and NDCG with respect to different window sizes. We choose the window size varied from 5 to 100, including 5, 10, 15, 20, 50 and 100. In the figures, the x-axis is the ranking list size k and the y-axis is the corresponding precision or NDCG. For simplicity, we only show the results of every ten ranking list size.

In Fig. 4(a), the precision of Word2API is stable when the window size is small. The performance is nearly the same for $w = 5$ and $w = 10$. For example, Precision@100 is 0.376 at $w = 5$ and 0.370 at $w = 10$. If we increase w to 50, the performance drops significantly. The reason may be that, Word2API constructs term vectors by maximizing the possibility to recover the current term vector with the co-occurred term vectors. As the window size increases, the difficulty of the training process also increases. For the CBOW model, the difficulty is caused by the averaging of the surrounding words, which dilutes most of the information in the training set. For the Skip-n model, the difficulty is caused by the need to find the relationship between the center word and every surrounding word in the increased window size.

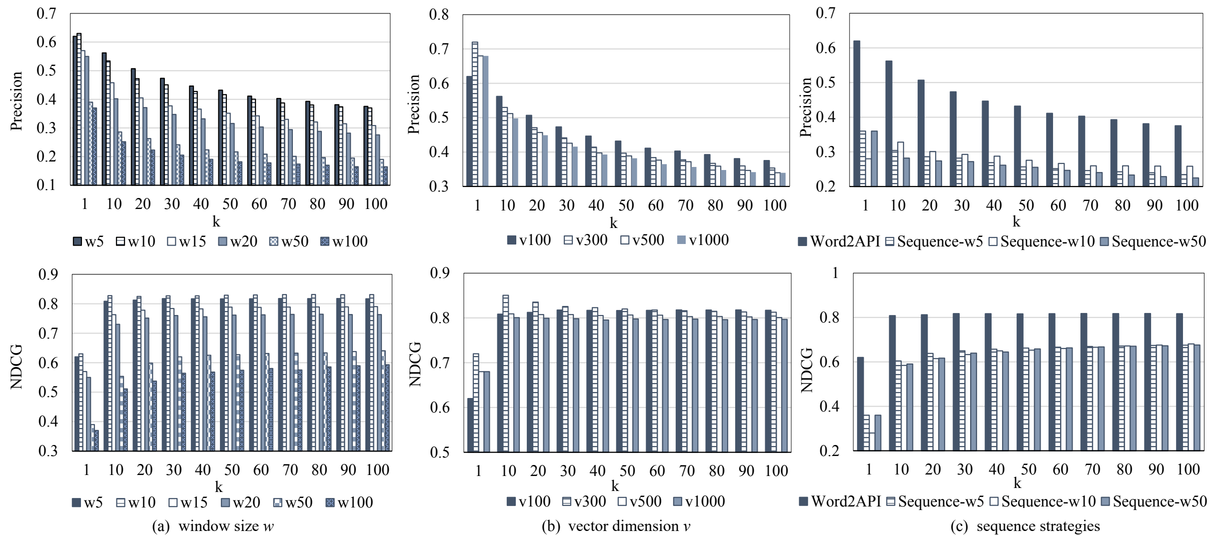
Similarly, NDCG also tends to be stable when the window size is small. We find that NDCG at $w = 10$ is consistently better than that at $w = 5$, which means we can further improve Word2API by tuning the parameters.

5.2.2 Vector Dimension

We evaluate the influence of vector dimensions in Fig. 4(b). The dimension is varied from 100 to 1000. For the top-1 result, the maximum margin of different dimensions is 0.100 on both precision and NDCG, which happens between $v = 100$ and $v = 300$. When Word2API recommends 100 APIs for a word, the variation becomes small. Precision@100 is 0.375 at $v = 100$ and 0.340 at $v = 1000$. NDCG@100 is 0.817 at $v = 100$ and 0.797 at $v = 1000$. We also average the differences between $v = 100$ and $v = 1000$ for the ranking list from 1 to 100. The average difference between $v = 100$ and $v = 1000$ is 0.050 for precision and 0.018

0098-5589 (c) 2018 IEEE. Personal use is permitted, but republication/redistribution requires IEEE permission. See http://www.ieee.org/publications_standards/publications/rights/index.html for more information.

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.2018.2876006, IEEE Transactions on Software Engineering



(a) window size w (b) vector dimension v (c) sequence strategies

Fig. 4: Precision and NDCG between Word2API and its variants. We tune the window size in (a). The vector dimension is evaluated in (b). We compare the shuffling strategy in Word2API with a sequence strategy in (c).

for NDCG. Hence, Word2API is relatively insensitive to the vector dimension overall.

The vector dimension determines the granularity to represent a term. A small vector dimension means to represent a term with some abstract entries, while a large vector dimension may generate more fine-grained vector representations. Although a large vector dimension may better represent words and APIs, it requires more data for training which slightly reduces Word2API's performance. Hence, the overall ability of Word2API is not significantly affected.

Conclusion. Word2API is stable at small window size and relatively insensitive to the vector dimension. We can improve Word2API by setting different parameters.

5.3 Answer to RQ3: The Shuffling Strategy

5.3.1 Comparison with the sequence strategy

Word2API constructs word-API tuples from method comments and API calls to train word embedding. It uses a shuffling strategy to obtain enough collocations between words and APIs in a word-API tuple. In this subsection, we compare the shuffling strategy against a sequence strategy. The sequence strategy combines the word sequence and the API sequence in a word-API tuple according to their original order, i.e., words come before the APIs. Then, it trains vectors on these combined data with the word embedding tool by the default parameters, namely $w = 5$, $v = 100$, and *-min-count*$=5$. We refer it as 'Sequence-w5'.

We compare Word2API and Sequence-w5 in Fig. 4(c). Sequence-w5 performs rather poor in estimating word-API relatedness. Both Precision@1 and NCDG@1 are 0.360. For top-100 recommended APIs, the precision and NDCG are 0.234 and 0.676 respectively, In contrast, Word2API significantly outperforms Sequence-w5 by up to 26% for both precision and NDCG. The results demonstrate that the shuffling strategy improves the ability of Word2API to construct vectors for semantically related words and APIs.

In addition, we increase the window size of Sequence-w5 to $w = 10$ and $w = 50$, denoted as 'Sequence-w10' and 'Sequence-w50'. The two variants investigate whether we can improve Sequence-w5 by increasing the window size. As shown in Fig. 4(c), Sequence-w10 and Sequence-w50 perform similar to Sequence-w5. For example, Precision@100 are 0.2344 and 0.2248, and NDCG@100 are 0.6755 and 0.6761 for Sequence-w5 and Sequence-w50 respectively. The differences are less than 0.01. The reason may be that, although a large window size increases the number of co-occurred words and APIs for training word embedding, it at the same time increases the difficulty of the training process as discussed in Section 5.2. These two factors result in a stable performance of the sequence strategy.

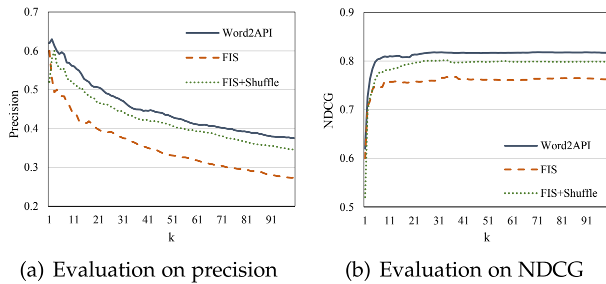
5.3.2 Comparison with the frequent itemset strategy

This subsection compares the shuffling strategy with an alternative strategy, namely the Frequent ItemSet (FIS) strategy, to generate a training set. FIS takes each word-API tuple as a document and mines frequent itemsets with the Apriori algorithm. To analyze the word-API relationship, we collect the frequent 2-itemsets that contain a word and an API. These word-API itemsets are considered to be highly related. We calculate the confidence value from the word to the API in the frequent 2-itemsets. After calculation, we traverse the 13,883,230 word-API tuples. For an API in a word-API tuple, we search its highly related words in the same tuple and put the API near the word with the largest word-to-API confidence value (on the right side of the word). If the highly related words are not found, we leave the API at its original position. We use these reordered word-API tuples to train word embedding.

There are two parameters for Apriori, i.e., the support value and the confidence value. The support value is set to 0.0001. We find each term in the word-API tuples appears in 1,491 tuples on average. We consider an itemset to be frequent when all the terms in the itemset appear more frequently than the average value, which attributes to a support value of 1,491/13,883,230, approximating to 0.0001. At last, 48,961 frequent 2-itemsets are mined. These itemsets

0098-5589 (c) 2018 IEEE. Personal use is permitted, but republication/redistribution requires IEEE permission. See http://www.ieee.org/publications_standards/publications/rights/index.html for more information.

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.2018.2876006, IEEE Transactions on Software Engineering



(a) Evaluation on precision (b) Evaluation on NDCG

Fig. 5: Comparison on Shuffling and FIS strategies.

contain 1,233 words. Each word is related to 40 APIs on average. We do not set a confidence value to further filter these itemsets, because when the number of frequent itemsets is small, most word-API tuples are kept as their original order.

As shown in Fig. 5, Word2API outperforms FIS by 0.02 to 0.102 in terms of precision and by 0.02 to 0.061 in terms of NDCG. Although FIS is useful to generate the training set, the shuffling strategy seems better than FIS. The reason is that there are valuable information among all words and APIs in the same tuple. When generating the training set with FIS, the word embedding algorithm mainly mines the information among the highly related words and APIs instead of the overall information.

To prove this assumption, we propose another strategy named FIS+Shuffle. This strategy first puts the highly related APIs near the word, and then shuffles the remaining words and APIs. In Fig. 5, FIS+Shuffle improves FIS. It means the shuffling strategy helps word embedding to analyze the overall information in a word-API tuple. However, Word2API still outperforms FIS+Shuffle. The reason may be that, for a word in frequent itemsets, word embedding can hardly find the relationship between this word and every API, as most surrounding APIs are limited to a few highly related ones.

Conclusion. The shuffling strategy improves the ability of Word2API to learn word-API relationships.

6 Word2API for API Sequences Recommendation

In Section 5, we evaluate Word2API on relatedness estimation at the word-API level. In the following parts, we further evaluate Word2API at the words-APIs level. We show two typical applications of Word2API, including API sequences recommendation and API documents linking.

6.1 Overview

The first application is API sequences recommendation. It helps developers find APIs related to a short natural language query. For example, if a developer searches for APIs implementing 'generate random number', APIs of 'Random#new, Random#nextInt' may be recommended.

For a recommendation system, recent studies show that API based query expansion is effective to search related APIs [3], [7]. Given a query, API based query expansion expands the query into an API vector. Each entry of the vector is the probability or similarity that an API is related to the query. The recommendation system uses the expanded API vector to search API sequences from a code base.

6.2 Approach: API based Query Expansion

We explain and compare the main algorithms for API based query expansion in this subsection, including word alignment expansion (Align$_{Exp}$), API description expansion (Des$_{Exp}$) and Word2API expansion (Word2API$_{Exp}$).

6.2.1 Word Alignment Expansion

Align$_{Exp}$ [3] uses a statistical word alignment model [42] to calculate the probability between an API and a query. The model is trained on alignment documents that consist of a set of words and related APIs [3]. We construct the alignment documents with word-API tuples [8]. We use GIZA++[16] to implement the word alignment model and transform the query into a vector based on the probabilities.

6.2.2 API Description Expansion

Lv et al. expand a user query by analyzing the API descriptions [7]. Given a query, Des$_{Exp}$ collects all APIs and their descriptions in the Java SE API specification. It calculates the similarity between the query and an API with a combined score of text similarity and name similarity. Text similarity measures the similarity between the query and an API description by cosine similarity in the Term Frequency and Inverted Document Frequency (TF-IDF) space. Name similarity splits an API into words by its camel style and measures the similarity between the query and the words. Des$_{Exp}$ transforms the query into an API vector by the combined score.

6.2.3 Word2API Expansion

Given a user query (a set of words) and an API, we apply the Words-APIs Similarity (formula 5) to calculate their similarity. For each API, formula 5 calculates the similarity between this API and each word in the query and then selects the largest value as the similarity between this API and the query. In this way, we can get similarity values between the query and every API in the Java SE APIs. Based on the similarities with all APIs, we follow the previous study [7] to select the top-10 APIs to expand the user query into an API vector. The length of the vector is 10. Each dimension of the vector represents an API. The value of the dimension is the similarity between this API and the query.

After query expansion, we employ a uniform framework to recommend API sequences [3]. This framework searches the word-API tuples to recommend APIs. It transforms the APIs in each word-API tuple into a 10-dimensional vector, in which each entry determines whether or not a selected top-10 API occurs in the current word-API tuple (0 or 1). Then, it ranks the word-API tuples according to the cosine similarity between the expanded API vector and every 0-1 vector. The framework finally returns the top-ranked word-API tuples. Each tuple contains a set of APIs. The order of these APIs is the same as that of in the word-API tuple. The framework is efficient and naive to highlight the effect of different expansion approaches.

16. GIZA++. http://www.statmt.org/moses/giza/GIZA++.html

0098-5589 (c) 2018 IEEE. Personal use is permitted, but republication/redistribution requires IEEE permission. See http://www.ieee.org/publications_standards/publications/rights/index.html for more information.

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.2018.2876006, IEEE Transactions on Software Engineering

For this application, the role of word2API is to calculate the similarity (relatedness) between a query and each API. The similarities are used to expand a query into an API vector for searching word-API tuples. We name this application as 'API sequences recommendation', because each word-API tuple corresponds to an API sequence. We find that word-API tuples not only have the APIs to implement a query, but also introduce the context or examples on using the APIs, as all word-API tuples are extracted from real-world source code. For example, for the API 'JFileChooser#showOpenDialog' which implements 'open file dialog', the word-API tuples usually contain APIs of 'JFileChooser#new' or 'JFileChooser#getSelectedFile'. These APIs provide examples on what to do before or after using 'JFileChooser#showOpenDialog'. Hence, comparing with other frameworks, e.g., deep neural network [8], a retrieval based framework recommends valid and real-world API sequences, that can be directly linked to diverse source code for understanding. We compare Word2API with a deep neural network framework in Sec. S7 of the supplement.

6.3 Evaluation: Query Expansion Algorithms

6.3.1 Motivation

We compare Word2API$_{Exp}$ with Align$_{Exp}$ and Des$_{Exp}$ in recommending Java SE API sequences.

6.3.2 Evaluation Method

First, we evaluate these algorithms with 30 human written queries [3], [8] listed in the first two columns of Table 3. The evaluation is quantified with First Rank (FR) and Precision@k [3]. FR is the position of the first related API sequence to a query and Precision@k is the ratio of related API sequences in the top-k results. Similar to the previous study [7], two authors examined the results. An API sequence is related if it contains the main API to implement a query and receives related feedback from both authors.

Second, we conduct an automatic evaluation [8] with 10,000 randomly selected tuples from all the word-API tuples. We treat the word sequences in these tuples as queries and the API sequences as the oracles. The queries are used for an algorithm to search API sequences in the remaining tuples. We compare sequence closeness between a recommended API sequence Seq_{rec} and the oracle sequence Seq_{orc} by BLEU score [43]:

$$BLEU = BP \cdot \exp\left(\sum_{n=1}^{N} w_n \log \frac{\#\text{n-grams in } Seq_{rec} \text{ and } Seq_{orc} + 1}{\#\text{n-grams in } Seq_{rec} + 1}\right)$$
$$BP = \begin{cases} 1 & \text{if } |Seq_{rec}| > |Seq_{orc}| \\ e^{1-|Seq_{orc}|/|Seq_{rec}|} & \text{if } |Seq_{rec}| \leq |Seq_{orc}| \end{cases} \quad (11)$$

where $|\cdot|$ is the length of a sequence, N is the maximum gram number and w_n is the weight of each type of gram. According to previous studies [8], [44], N is set to 4 and $w_n = 1/N$. It means that we calculate the overlaps of n-grams of Seq_{rec} and Seq_{orc} from 1 to 4 with equal weights.

For a ranking list of k API sequences, the BLEU score of the list is the maximum BLEU score between Seq_{rec} and Seq_{orc} [8]. Since 10,000 tuples are used for evaluation, we remove these tuples and their duplicate copies from the word-API tuples to re-train Word2API for fair comparison.

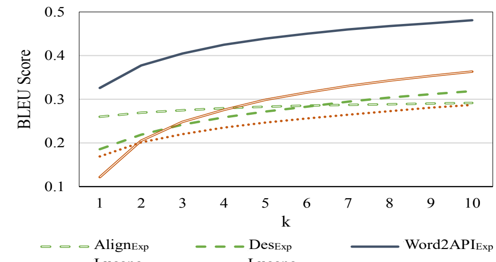


Fig. 6: BLEU score for different expansion algorithms.

6.3.3 Result

Table 3 shows the results for 30 human written queries. 'NF' means Not Found related APIs in the top 10 results. We treat the FR value of 'NF' as 11 to calculate the average FR [8]. For 9 out of the 30 queries, all the query expansion algorithms can recommend related API sequences at top-1. However, Align$_{Exp}$ fails to recommend APIs for many queries. The reason is that, when Align$_{Exp}$ expands the query into an API vector, the top ranked APIs in the vector are unrelated to the correct APIs.

The average FR of Word2API$_{Exp}$ is 1.933. A related result is ranked at top-1 for 20 out of the 30 queries. For precision, the average top-5 and top-10 precision by Word2API$_{Exp}$ is 0.680 and 0.677 respectively. The results are superior to those of Align$_{Exp}$ and Des$_{Exp}$, whose average top-10 precision values are 0.463 and 0.533 respectively. Hence, by expanding queries with Word2API$_{Exp}$, the recommendation framework achieves more related results on average than Align$_{Exp}$ and Des$_{Exp}$. We conduct the paired Wilcoxon signed rank test on the 30 queries in the last row of Table 3. When comparing Word2API against Align$_{Exp}$ and Des$_{Exp}$, the p-values of FR, Precision@5 and Precision@10 are 0.0023, 0.1462, 0.0408, and 0.0117, 0.1347, 0.0675 respectively.

We report the results under 10,000 constructed queries in Fig. 6. For the expansion approaches, Word2API$_{Exp}$ shows the best ability to transform queries into API vectors. The BLEU scores are between 0.326 and 0.481, which outperform Align$_{Exp}$ and Des$_{Exp}$ by up to 0.140 in terms of BLEU@1 and 0.189 in terms of BLEU@10. The results pass the Wilcoxon test with p-value$<$0.025 after Bonferroni correction. Since we take the same naive framework to retrieve API sequences, it demonstrates that Word2API$_{Exp}$ makes the key contribution to the results.

Table 4 shows the top-5 expanded APIs by Word2API corresponding to each human written query. In contrast to Table 2, this table reflects the ability of Word2API in understanding a set of words instead of a single one. Among the top ranked APIs, most APIs are directly related to the queries. For example, Word2API finds APIs of 'Pattern#compile, Pattern#pattern, Pattern#matcher' for query Q10 'match regular expressions'. APIs of 'Random#nextInt, Random#nextDouble, Random#nextBytes' are ranked high for query Q12 'generate random number'. With these APIs, the naive framework can find more related API sequences on average and rank the first related API sequence (FR) higher than the comparison algorithms.

Besides, we find that Word2API can interpret a query with APIs from multiple classes. For example, in

0098-5589 (c) 2018 IEEE. Personal use is permitted, but republication/redistribution requires IEEE permission. See http://www.ieee.org/publications_standards/publications/rights/index.html for more information.

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.2018.2876006, IEEE Transactions on Software Engineering

TABLE 3: Performance over 30 human written queries. P is short for Precision.

ID	Query	Query Expansion									Search Engine								
		$Align_{Exp}$ [3]			Des_{Exp} [7]			$Word2API_{Exp}$			$Google_{GitHub}$			$Lucene_{API}$			$Lucene_{API+Comment}$		
		FR	P@5	P@10	FR	P@5	P@10	FR	P@5	P@10	FR	P@5	P@10	FR	P@5	P@10	FR	P@5	P@10
Q1	convert int to string	NF	0	0	NF	0	0	3	0.2	0.1	6	0	0.1	NF	0	0	1	0.4	0.2
Q2	convert string to int	1	1	0.5	NF	0	0	1	0.8	0.8	1	0.8	0.7	NF	0	0	8	0	0.3
Q3	append string	1	1	1	1	1	1	1	1	1	7	0	0.2	1	1	1	1	1	1
Q4	get current time	NF	0	0	NF	0	0	1	1	1	1	0.8	0.5	1	1	1	1	1	0.8
Q5	parse datetime from string	10	0	0.1	NF	0	0	1	1	0.7	1	1	1	NF	0	0	NF	0	0
Q6	test file exists	1	1	1	1	1	1	1	0.8	0.8	2	0.8	0.9	NF	0	0	NF	0.2	0.1
Q7	open a url	1	1	1	1	1	1	1	0.8	0.8	1	0.2	0.3	1	1	1	1	1	1
Q8	open file dialog	NF	0	0	1	0.8	0.7	1	0.4	0.7	1	0.6	0.5	1	1	1	3	0.2	0.3
Q9	get files in folder	NF	0	0	1	0.8	0.9	1	1	0.9	1	0.6	0.5	NF	0	0	NF	0	0
Q10	match regular expressions	1	1	0.8	1	0.6	0.7	1	1	1	2	0.2	0.5	NF	0	0	1	1	0.9
Q11	generate md5 hash code	NF	0	0	NF	0	0	1	1	1	1	0.8	0.6	NF	0	0	8	0	0.2
Q12	generate random number	1	0.4	0.2	1	1	1	1	1	1	1	0.8	0.6	1	0.6	0.6	1	1	0.9
Q13	round a decimal value	NF	0	0	2	0.2	0.1	1	0.8	0.8	10	0	0.1	5	0.2	0.6	1	0.8	0.8
Q14	execute sql statement	NF	0	0	NF	0	0	2	0.6	0.5	1	1	0.8	1	0.8	0.9	1	0.8	0.7
Q15	connect to database	1	1	1	NF	0	0	1	1	1	1	0.8	0.6	NF	0	0	NF	0	0
Q16	create file	1	1	1	1	1	1	1	1	1	1	0.6	0.7	NF	0	0	1	0.4	0.2
Q17	copy file	1	1	1	1	1	1	1	0.6	0.5	1	0.8	0.6	NF	0	0	4	0.2	0.1
Q18	copy a file and save it to your destination path	1	1	1	2	0.2	0.3	1	0.8	0.9	7	0	0.2	NF	0	0	10	0	0.1
Q19	delete files and folders in a directory	1	1	1	3	0.6	0.4	4	0.4	0.4	4	0.4	0.5	NF	0	0	1	0.8	0.4
Q20	reverse a string	NF	0	0	NF	0	0	NF	0	0	1	1	1	NF	0	0	5	0.2	0.1
Q21	create socket	NF	0	0	1	0.6	0.4	1	1	0.9	4	0.2	0.3	1	1	0.7	NF	0	0
Q22	rename a file	NF	0	0	NF	0	0	4	0.4	0.5	1	0.6	0.3	NF	0	0	NF	0	0
Q23	download file from url	1	1	0.7	1	1	1	5	0.2	0.3	1	1	0.7	9	0	0.2	5	0.2	0.5
Q24	serialize an object	1	1	1	1	1	1	1	1	1	3	0.2	0.3	4	0.4	0.2	1	0.6	0.3
Q25	read binary file	1	1	0.6	1	1	1	1	0.8	0.8	2	0.4	0.4	7	0	0.1	2	0.6	0.7
Q26	save an image to a file	1	1	1	1	1	1	5	0.2	0.4	1	0.4	0.4	1	1	1	4	0.2	0.6
Q27	write an image to a file	1	1	1	1	0.8	0.6	2	0.4	0.3	1	0.8	0.8	1	1	1	1	1	1
Q28	parse xml	NF	0	0	NF	0	0	1	0.2	0.3	1	0.6	0.6	5	0.2	0.1	1	0.2	0.1
Q29	play audio	NF	0	0	1	0.8	0.9	1	0.4	0.5	1	0.6	0.6	6	0	0.2	1	0.2	0.2
Q30	play the audio clip at the specified absolute URL	NF	0	0	1	1	1	1	0.6	0.4	1	1	0.8	4	0.4	0.7	2	0.8	0.9
Avg.		5.633	0.513	0.463	4.467	0.547	0.533	**1.933**	**0.68**	**0.677**	2.233	0.567	0.537	6.767	0.32	0.343	4.367	0.427	0.413
p		0.002	0.146	0.04	0.01	0.135	0.068	*	*	*	0.568	0.109	0.02	<0.001	0.002	0.003	0.014	0.009	0.006

TABLE 4: Top-5 expanded APIs ranked by Word2API over 30 human written queries.

Ranked APIs	Ranked APIs	Ranked APIs	Ranked APIs	Ranked APIs
Q1	Q2	Q3	Q4	Q5
NFException[1]#toString	NFException[1]#toString	StringBuffer#append	System#currentTimeMillis	SimpleDateFormat#parse
Integer#parseInt	Integer#parseInt	StringBuilder#append	Date#getTime	DateFormat#parse
Object#toString	Object#toString	StringBuffer#toString	Clock#millis	GCalendar[2]#toZonedDateTime
Byte#parseByte	Byte#parseByte	StringBuilder#toString	Calendar#getTimeInMillis	Date#getField
Integer#byteValue	Integer#byteValue	StringBuffer#length	BeanContext#isDesignTime	Calendar#clear
Q6	Q7	Q8	Q9	Q10
File#exists	URL#toString	JFileChooser#getSelectedFile	File#getName	Pattern#flags
File#isFile	URL#openStream	JFileChooser#showOpenDialog	File#getParentFile	Pattern#compile
File#canRead	URL#openConnection	JFileChooser#setDialogTitle	File#getPath	Pattern#pattern
File#isDirectory	URLConnection#getInputStream	JFileChooser#setCurrentDirectory	File#isFile	Matcher#toMatchResult
File#getPath	URL#toExternalForm	JFChooser[3]#setFileSelectionMode	File#getAbsoluteFile	Pattern#matcher
Q11	Q12	Q13	Q14	Q15
MessageDigest#digest	SecureRandom#nextInt	Math#round	Statement#close	DataSource#getConnection
MessageDigest#getInstance	Random#nextInt	BigDecimal#toPlainString	Connection#createStatement	Connection#close
TreeMap#hashCode	Random#nextDouble	BigDecimal#movePointRight	SQLException#getMessage	Connection#isClosed
InvocationHandler#hashCode	Random#nextBytes	BigDecimal#compactLong	Connection#close	DriverManager#getConnection
NSAException[4]#printStackTrace	SecureRandom#nextBytes	Math#floor	Connection#prepareStatement	Connection#getNetworkTimeout
Q16	Q17	Q18	Q19	Q20
File#exists	File#toPath	File#getParentFile	File#isDirectory	StringBuilder#reverse
File#toPath	Files#copy	File#mkdirs	File#exists	Collections#reverse
File#getAbsoluteFile	File#mkdirs	File#toPath	File#isFile	StringBuffer#reverse
File#getParentFile	Arrays#copyOf	File#getPath	File#getParentFile	Collections#sort
Path#toFile	PFPermissions[5]#asFileAttribute	Files#copy	File#getPath	Collections#reverseOrder
Q21	Q22	Q23	Q24	Q25
SocketFactory#getClass	File#renameTo	URL#toURI	OutputStream#hashCode	DataInputStream#close
SSLSocket#connect	File#delete	URL#toString	OOStream[6]#dumpElementln	BufferedInputStream#read
ServerSocket#getChannel	File#getParentFile	URL#getFile	Component#doSwingSerialization	File#length
SSLSocketFactory#getDefault	File#exists	URLConnection#getContentLength	Externalizable#getClass	FileInputStream#close
ServerSocketChannel#setOption	File#getName	URL#getPath	ObjectOutputStream#writeObject	File#canRead
Q26	Q27	Q28	Q29	Q30
ImageIO#write	ImageIO#write	DocumentBuilder#parse	AudioSystem#getLine	AudioSystem#getAudioInputStream
File#exists	FileOutputStream#close	DBFactory[7]#newDocumentBuilder	LUException[8]#printStackTrace	Clip#start
FileOutputStream#close	File#createTempFile	DBFactory[7]#newInstance	SourceDataLine#start	Clip#open
File#getPath	File#mkdirs	Document#getDocumentElement	Clip#start	AudioSystem#getClip
ImageIO#read	OutputStream#close	Element#getAttribute	AudioSystem#getAudioInputStream	Clip#stop

[1] NFException: NumberFormatException [2] GCalendar: GregorianCalendar [3] JFChooser: JFileChooser [4] NSAException: NoSuchAlgorithmException
[5] PFPermissions: PosixFilePermissions [6] OOStream: ObjectOutputStream [7] DBFactory: DocumentBuilderFactory [8] LUException: LineUnavailableException

query Q4 'get current time', the top ranked APIs are 'System#currentTimeMillis', 'Date#getTime', and 'Calendar#getTimeInMillis'. These APIs belong to different classes, including 'java.lang.System', 'java.util.Date', and 'java.time.Clock'. The same phenomenon can also be found in other queries, e.g., Q3 'append string', Q15 'connect to database', etc. It means that Word2API may help developers understand a query by providing diverse APIs.

6.3.4 Conclusion

With the Word2API-expanded query, a system for API sequences recommendation significantly outperforms the comparison ones in terms of FR and BLEU score.

6.4 Evaluation: General-purpose Search Engines

6.4.1 Motivation

This section evaluates the performance of general-purpose search engines on recommending query-related APIs.

0098-5589 (c) 2018 IEEE. Personal use is permitted, but republication/redistribution requires IEEE permission. See http://www.ieee.org/publications_standards/publications/rights/index.html for more information.

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.2018.2876006, IEEE Transactions on Software Engineering

6.4.2 Evaluation Method

We propose three search engine based methods

Google$_{GitHub}$. This method searches a user query with Google search engine and collects the APIs in the top-10 web pages as results. Since this study uses Java projects on GitHub to evaluate the ability of algorithms to match user queries with APIs, for fair comparison, we limit Google$_{GitHub}$ to search the resources on GitHub by rewriting a query as 'java *query* site:github.com'.

Lucene$_{API}$. The second method uses Lucene to search API sequences. Lucene is a widely used open-source search engine that uses text matching on words in queries and target documents to find related documents [45]. Lucene$_{API}$ takes the API sequences in word-API tuples as documents. We first split each API in a word-API tuple into a set of words by its camel style. Then, stop words removal and stemming are performed on the split words. After that, we index these words as a document for search.

Lucene$_{API+Comment}$. The third method provides more knowledge to Lucene for accurate search. Besides the words in API sequences, Lucene$_{API+Comment}$ also indexes the words in the word sequence of a word-API tuple. Since Word2API also uses API calls and method comments to mine word-API relationships, this method helps us understand whether general-purpose search engines can better mine the semantic relatedness when provided with the same amount of information.

We use the 30 human queries and 10,000 automatically constructed queries for evaluation. We do not evaluate Google$_{GitHub}$ with the 10,000 queries due to the network problem of automatically sending 10,000 queries to Google. Since Google$_{GitHub}$ only returns web pages instead of Java APIs, we use the following principle to evaluate Google$_{GitHub}$. We label a web page as correct, if the web page:

- contains APIs related to the query, even though the APIs are from non-core Java APIs or other programming languages, e.g., Groovy and Scala;
- does not contain related APIs, but it implements a new method related to the query;
- does not contain source code, e.g., issue reports, but it contains API-like words related to the query.

6.4.3 Result

In Table 3, the average FR, P@5 and P@10 of Word2API$_{Exp}$ are superior to those of Google$_{GitHub}$. When considering the p-values, Word2API performs similar to Google$_{GitHub}$ in terms of FR and P@5, but significantly outperforms Google$_{GitHub}$ in terms of P@10 ($p<0.05$). We find that even though Google$_{GitHub}$ is limited to search GitHub resources, the correct APIs for some queries can still be easily obtained by matching a query with web page titles. Hence, the results of Google$_{GitHub}$ may be attributed to both understanding the word-API relationships and leveraging the tremendous knowledge on the Internet. Since Word2API$_{Exp}$ does not aim to recommend APIs with all the knowledge on the Internet, we conclude that Word2API$_{Exp}$ achieves similar or better results compared to Google$_{GitHub}$ by only analyzing word-API relatedness.

For Lucene based methods, Word2API$_{Exp}$ significantly outperforms Lucene$_{API}$ in terms of FR, P@5 and P@10. It means the semantic gaps between words and APIs hinder the performance of general-purpose search engines in searching APIs by words. When we provide more knowledge for Lucene, i.e., both API calls and method comments, the performance of Lucene$_{API+Comment}$ improves. However, Word2API$_{Exp}$ still outperforms Lucene$_{API+Comment}$. Since Word2API$_{Exp}$ and Lucene$_{API+Comment}$ use the same information to mine word-API relationships, it means Word2API can better mine the semantic relatedness compared to a general-purpose search engine Lucene in this case, when provided with the same amount of knowledge.

6.4.4 Conclusion

The semantic gaps hinder the performance of search engines in understanding APIs. Word2API analyzes word-API relationships better than the search engine Lucene when provided with the same amount of knowledge.

7 Word2API for API Documents Linking

7.1 Overview

The second application is API documents linking [4] which analyzes the relationships between API documents and the questions in Q&A (Question & Answer) communities, e.g., Stack Overflow. This application is more complex, since it needs to estimate semantic relatedness between a set of words and APIs, instead of a single API each time.

In Q&A communities, participators discuss technical questions by replying and scoring to each other. Given a newly submitted question, participators usually discuss and comprehend it with some APIs. A statistic shows that more than one third (38%) answers in Stack Overflow have at least one API [46]. Hence, linking API documents to newly submitted questions may save participators' time to answer the questions [4]. In this part, we link questions in Stack Overflow to the documents in Java SE API specification [4].

7.2 Approach: API Documents Linking

Give a newly submitted question, we introduce four typical algorithms to recommend related API documents.

7.2.1 Vector Space Model (VSM)

VSM transforms the question and API documents into vectors, in which each entry is a word weighted by the TF-IDF strategy. Then it ranks API documents by calculating the cosine similarity between the question vector and API document vectors. We split the APIs and API-like words in these texts by the camel style to increase the number of matched words.

7.2.2 Standard Word Embedding (WE)

Ye et al. train word vectors for relatedness estimation with a standard word embedding model [4]. The vectors are generated by analyzing the words in Java and Eclipse API specifications, user/developer guides, and tutorials. To link a question with API documents, they transform the question and each API document into two words sets. Then, they calculate the similarities of the word sets with the word vectors in a similar way as Formula 5 (Words-APIs Similarity), which replaces the word set and API set in Formula 5 with

0098-5589 (c) 2018 IEEE. Personal use is permitted, but republication/redistribution requires IEEE permission. See http://www.ieee.org/publications_standards/publications/rights/index.html for more information.

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.2018.2876006, IEEE Transactions on Software Engineering

two word sets. For fair comparison, we also add the word sequences in word-API tuples for training. WE is trained by the default parameters of the word embedding tool.

7.2.3 Word2API Approach (Word2API)

Word2API first extracts the words from the question and the method level APIs of an API type from each API document. Then, it calculate the relatedness between the word set and API set by the Words-APIs Similarity.

7.2.4 Integrated Approaches

Previous studies also integrate VSM and WE to generate an integrate approach [4]. Given a question and an API document, we denote the similarity calculated by VSM, WE and Word2API as Sim_{VSM}, Sim_{WE}, and Sim_{Word2API} respectively. We rank API documents by two types of integrations, namely VSM-WE ($Sim_{\text{VSM-WE}}$) [4] and VSM-Word2API ($Sim_{\text{VSM-Word2API}}$).

$$Sim_{\text{VSM-WE}} = \alpha \times Sim_{\text{VSM}} + (1-\alpha) \times Sim_{\text{WE}}, \quad (12)$$

$$Sim_{\text{VSM-Word2API}} = \alpha \times Sim_{\text{VSM}} + (1-\alpha) \times Sim_{\text{Word2API}}, \quad (13)$$

where α is the weight of different approaches. The values are 0.18 and 0.36 for $Sim_{\text{VSM-WE}}$ and $Sim_{\text{VSM-Word2API}}$ respectively as we will discuss later.

7.3 Evaluation

7.3.1 Motivation

We evaluate the effectiveness of Word2API against the comparison algorithms on API documents linking.

7.3.2 Evaluation Method

We follow Ye et al. [4] to construct a benchmark for evaluation. We download Java tagged questions in Stack Overflow between August 2008 and March 2014, since these questions have stabilized, i.e., no more edits are likely to be done. Then we select a question, if the score of the question exceeds 20, the score of its 'best/accepted' answer exceeds 10, and the 'best/accepted' answer has at least one link to the Java SE API specification [4]. According to the criteria, 555 questions are collected. We partition these questions into two parts. The first 277 questions form a training set and the latter part is a testing set. The size of the testing set is similar to the previous study [4]. We use the training set to tune the parameter α of the integrated approaches. For an approach, we traverse α from 0.01 to 1.0 with a stepwise 0.01 and take the value that maximizes MAP in Equ. 14 as the final parameter value. We take the API documents linked in the best/accepted answer as the oracle for evaluation.

For the testing set, we compare the oracle API documents and the top-10 recommended API documents by MAP and MRR [4]. MAP is the mean of the average precision for each question.

$$\begin{aligned} MAP &= \frac{1}{|Q|}\sum_{i=1}^{Q} AvgP_i \\ AvgP &= \sum_{k=1}^{N} r_k * Precision@k \end{aligned} \quad (14)$$

TABLE 5: MAP and MRR comparison.

Algorithms	MAP	MRR
VSM	0.232	0.259
WE [4]	0.313	0.354
Word2API	0.402	0.433
VSM+WE [4]	0.340	0.380
VSM+Word2API	0.436	0.469

where N is the number of recommended API documents for a question, Precision@k is the ratio of correctly recommended API documents in the top-k results, and r is a flag that r_k= 1 if the kth result is correct and r_k= 0 otherwise.

MRR is the mean reciprocal rank of the first correctly recommended API document for each question.

$$MRR = \frac{1}{|Q|}\sum_{i=1}^{Q}\frac{1}{FR_i}, \quad (15)$$

where $|Q|$ is the number of questions in the testing set and FR_i is the position of the first related API document for Q_i.

7.3.3 Result

Table 5 presents MAP and MRR of different algorithms. Among the three atomic algorithms, the embedding based algorithms (WE and Word2API) are superior to VSM. They improve VSM by up to 0.170 and 0.174 over MAP and MRR respectively. The results show that sematic relatedness calculated by word embedding based algorithms are better than simple text matching (VSM) for this task. For the two embedding based algorithms, Word2API performs better. The results of MAP and MRR for Word2API are 0.402 and 0.433 over the testing set, which outperform WE by 0.089 and 0.079 respectively. It means that Word2API is more effective in mining semantic relatedness between words and APIs than WE, which treats APIs as words.

We also find that text matching based algorithm VSM and embedding based algorithms can reinforce each other, since they measure documents from different perspectives. When we integrate the two types of algorithms, the results have an improvement by around 3%, e.g., VSM+Word2API reaches 0.436 on MAP and 0.469 on MRR.

Additionally, we note that some fine-grained text analysis techniques may further improve API documents linking, e.g., deducing the APIs in source code snippets of the questions [34], [47]. We discuss this observation in Sec. S9 of the supplement. The fine-grained analysis further improves API documents linking by nearly 5%.

7.3.4 Conclusion

Word2API is superior to VSM and WE on relatedness estimation for API documents linking.

8 THREATS TO VALIDITY

Construction Validity. Word2API may require a large number of word-API tuples to construct a model. As a machine learning algorithm, Word2API is trained on the historical knowledge of word-API relationships. When there are only a few word-API tuples containing an API, Word2API may not well learn the relationship between words and this API.

0098-5589 (c) 2018 IEEE. Personal use is permitted, but republication/redistribution requires IEEE permission. See http://www.ieee.org/publications_standards/publications/rights/index.html for more information.

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.2018.2876006, IEEE Transactions on Software Engineering

TABLE 6: Overview of the related work.

	Type	Paper	Knowledge
Semantic Rel. Estimation	Application	Gu et al. [8]	API sequences recommendation
		Ye et al. [4]	API documents linking
		Corley et al. [48]	Feature location
	Rule-based	Beyer et al. [49]	Heuristic rules
		Howard et al. [23]	Part-Of-Speech
		Yang et al. [50]	Term Morphology
	Corpus-based	Mahmoud et al. [11]	LSA, PMI, NSD (*RQ1*)
		Tian et al. [32], [51]	HAL (*RQ1*)
		Chen et al. [33]	Word embedding (*APP2*)
		Ye et al. [4]	Word embedding (*APP2*)
		Nguyen et al. [14]	API embedding (*Section S2 of the supplement*)
Query Expansion	Word-based	Wang et al. [52]	Relevance feedback
		Hill et al. [53], [54]	Co-occurred words
		Lu et al. [55]	WordNet
		Nie et al. [1]	Stack Overflow
		Campbell et al. [56]	Stack Overflow
	API-based	Lv et al. [7]	Similarity with API des. (*APP1*)
		Raghothaman et al. [3]	Word alignment (*APP1*)

A deep analysis is conducted in Sec. S8 of the supplement. However, as the prevalence of open source, we can easily download thousands of source code containing specific APIs from code repositories, e.g., GitHub, Google Code, etc. As our preliminary statistic on GitHub, more than 583,779 and 388,300 projects contains at least one Android API and C# API respectively. These projects may facilitate the training of Word2API for such target APIs.

In addition, there are also threats in the two applications of Word2API. We automatically select 10,000 word sequences to evaluate API sequences recommendation. Since word sequences in the comments are not exactly the same with human queries, we also evaluate Word2API with 30 human written queries.

External Validity. The first threat comes from the human judgement processes. To evaluate the semantic relatedness between query words and APIs, several human judgements are conducted. The selected query words may be vague for evaluation or unrealistic in real scenarios. Meanwhile, the judgements are subjective and may bring biases. We have observed some mislabeled APIs in this process. To alleviate biases, we follow the TREC strategy for human judgements. A re-evaluation shows a substantially agreement on the judgements with the Kappa score of 0.636. In addition, we share the human judgement results at https://github.com/softw-lab/word2api for research.

The second threat is the generality of Word2API. In this study, we evaluate Word2API at the word-API level with 50 query words and at the words-APIs level with two applications. More applications need to be investigated in the future. For generality, we only use the default parameters to train Word2API. Experiments show that Word2API works well without a fine-grained parameter optimization.

9 Related Work

We summarize the related work in Table 6, including semantic relatedness estimation and query expansion. For the highly related works, we also mark the *RQ* or *APP*lication that we compare these algorithms.

9.1 Semantic Relatedness Estimation

Semantic gaps between words and APIs negatively affect many software engineering tasks, e.g., API sequences recommendation [8], API documents linking [4], feature location [48], etc. In this study, we propose Word2API to analyze the relatedness between words and APIs in a fine-grained, task-independent way. Such analysis is useful for developers to understand the APIs and source code.

In the field of fine-grained relatedness estimation, Beyer et al. [49] propose nine heuristic rules to suggest synonyms for Stack Overflow tags. Howard et al. [23] and Yang et al. [50] infer software-based semantically-similar words by comparing the part-of-speech (verbs and nouns) and common words in API names and comments. These techniques rely on specific rules without analyzing word relationships.

Hence corpus-based methods are proposed. Mahmoud et al. [11] find that corpus-based methods outperform other methods on relatedness estimation. Tian et al. [32], [51] leverage Hyperspace Analogue to Language (HAL) to construct word vectors. Chen et al. [33] utilize word embedding to infer software-specific morphological forms, e.g., Visual C++ and VC++. Similar vectors are also constructed on Java/Eclipse tutorials and user guides [4]. Besides, Nguyen et al. propose API embedding to represent APIs of different languages [14].

Word2API is a corpus-based method. It outperforms previous algorithms in word-API relatedness estimation.

9.2 Query Expansion

We take code search and API sequences recommendation as representative examples to enumerate the work in query expansion. Code search aims to return code snippets for a user query [57]. These snippets are usually more domain specific than API sequences [8]. In this study, we classify query expansion into word-based expansion and API-based expansion.

Word-based expansion transforms a natural language query into more meaningful words. Wang et al. [52] leverage relevance feedback to expand queries with words in manually selected documents. Hill et al. [53], [54] expand a query with frequently co-occurred words in code snippets. Beside, external knowledge is also important for query expansion. Lu et al. [55] reformulate a user query with synonyms generated from WordNet. Code snippets from Stack Overflow are also used for expanding queries [1], [56]. However, only a small part of Stack Overflow questions contains complete code snippets [46]. In addition, word-based expansion aims at enhancing poor or simple queries. Yet, the gaps between natural languages and APIs still exist.

Therefore, recent studies propose API-based expansion to transform a user query into related APIs. Lv et al. [7] expand a query by the text similarity and the name similarity between the query and API descriptions. The effectiveness of this algorithms largely depends on the quality of API descriptions. Hence, Raghothaman et al. [3] utilize statistical word alignment models to expand queries into APIs.

Word2API belongs to API-based expansion. A comparison with previous studies shows that Word2API is effective in expanding queries into API vectors. In the future, we plan to conduct a comprehensive comparison and investigate the synergy of different types of expansion algorithms.

0098-5589 (c) 2018 IEEE. Personal use is permitted, but republication/redistribution requires IEEE permission. See http://www.ieee.org/publications_standards/publications/rights/index.html for more information.

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.2018.2876006, IEEE Transactions on Software Engineering

10 Conclusion and Future Work

In this study, we present our attempt towards constructing low-dimensional representations for both words and APIs. Our algorithm Word2API leverages method comments and API calls to analyze semantic relatedness between words and APIs. Experiments show that Word2API is effective in estimating semantically related APIs for a given word. We present two applications of Word2API. Word2API is a promising approach for expanding user queries into APIs and link API documents to Stack Overflow questions. In the future, we plan to employ Word2API for other programming languages and applications, and investigate different functions to measure similarity in addition to Words-APIs Similarity used in this paper.

Acknowledgment

We thank the volunteers for their contributions to the exhausted human judgements processes. We thank the reviewers for their insightful comments to improve this paper. Their comments help us look deep into Word2API. This work is supported by the National Key Research and Development Program of China under Grants 2018YFB1003900, and supported in part by the National Natural Science Foundation of China under Grants No. 61722202 and the JSPS KAKENHI Grant Number JP15H05306 and JP18H03222.

References

[1] L. Nie, H. Jiang, Z. Ren, Z. Sun, and X. Li, "Query expansion based on crowd knowledge for code search," *IEEE Trans. on Services Computing*, vol. 9, no. 5, pp. 771–783, 2016.

[2] M. P. Robillard and R. Deline, "A field study of API learning obstacles," *Empirical Software Engineering*, vol. 16, no. 6, pp. 703–732, 2011.

[3] M. Raghothaman, Y. Wei, and Y. Hamadi, "SWIM: synthesizing what I mean - code search and idiomatic snippet synthesis," in *Proc. of the 38th Int'l Conf. on Softw. Eng.* ACM, 2016, pp. 357–367.

[4] 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 *Proc. of the 38th Int'l Conf. on Softw. Eng.* ACM, 2016, pp. 404–415.

[5] R. A. Baeza-Yates and B. A. Ribeiro-Neto, *Modern Information Retrieval the Concepts and Technology Behind Search*. ACM Press Books, 2011.

[6] H. Jiang, X. Li, Z. Yang, and J. Xuan, "What causes my test alarm? automatic cause analysis for test alarms in system and integration testing," in *Proc. of the 39th Int'l. Conf. on Softw. Eng.* IEEE, 2017, pp. 712–723.

[7] F. Lv, H. Zhang, J.-g. Lou, S. Wang, D. Zhang, and J. Zhao, "Codehow: Effective code search based on API understanding and extended boolean model," in *30th IEEE/ACM Int'l Conf. on Automated Softw. Eng.* IEEE, 2015, pp. 260–270.

[8] X. Gu, H. Zhang, D. Zhang, and S. Kim, "Deep API learning," in *Proc. of the 2016 24th ACM SIGSOFT Int'l Symposium on Foundations of Softw. Eng.* ACM, 2016, pp. 631–642.

[9] M. Piccioni, C. A. Furia, and B. Meyer, "An empirical study of API usability," in *ACM / IEEE Int'l Symposium on Empirical Softw. Eng. and Measurement*, 2013, pp. 5–14.

[10] Y. Zhou, R. Gu, T. Chen, Z. Huang, S. Panichella, and H. Gall, "Analyzing APIs documentation and code to detect directive defects," in *Proc. of the 39th Int'l Conf. on Softw. Eng.*, 2017, pp. 27–37.

[11] A. Mahmoud and G. Bradshaw, "Estimating semantic relatedness in source code," *ACM Trans. on Softw. Eng. and Methodology*, vol. 25, no. 1, p. 10, 2015.

[12] T. K. Landauer and S. T. Dumais, "A solution to plato's problem: The latent semantic analysis theory of acquisition, induction, and representation of knowledge." *Psychological Review*, vol. 104, no. 2, p. 211, 1997.

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

[14] T. D. Nguyen, A. T. Nguyen, H. D. Phan, and T. N. Nguyen, "Exploring API embedding for API usages and applications," in *Proc. of the 39th Int'l. Conf. on Softw. Eng.* IEEE Press, 2017, pp. 438–449.

[15] Wikipedia, "Application programming interface," https://en.wikipedia.org/wiki/Application_programming_interface, 2017.

[16] J. Bloch, "How to design a good API and why it matters," in *Companion to the ACM Sigplan Symposium on Object-Oriented Programming Systems, Languages, and Applications*, 2006, pp. 506–507.

[17] T. Mikolov, I. Sutskever, K. Chen, G. Corrado, and J. Dean, "Distributed representations of words and phrases and their compositionality," *Advances in Neural Information Processing Systems*, vol. 26, pp. 3111–3119, 2013.

[18] T. Mikolov, K. Chen, G. Corrado, and J. Dean, "Efficient estimation of word representations in vector space," *CoRR, abs/1301.3781*, 2013.

[19] E. Kalliamvakou, G. Gousios, K. Blincoe, L. Singer, D. M. German, and D. Damian, "An in-depth study of the promises and perils of mining github," *Empirical Software Engineering*, vol. 21, no. 5, pp. 2035–2071, 2016.

[20] Oracle, "Overriding and hiding methods," http://docs.oracle.com/javase/tutorial/java/IandI/override.html, 2017.

[21] L. Pascarella and A. Bacchelli, "Classifying code comments in Java open-source software systems," in *Proc. of the 14th Int'l Conf. on Mining Software Repositories*. IEEE Press, 2017, pp. 227–237.

[22] S. Margaret-Anne, J. Ryall, R. I. Bull, D. Myers, and J. Singer, "ToDo or to bug: Exploring how task annotations play a role in the work practices of software developers," in *Proc. of the 30th Int'l. Conf. on Softw. Eng.*, pp. 251–260.

[23] M. J. Howard, S. Gupta, L. Pollock, and K. Vijay-Shanker, "Automatically mining software-based, semantically-similar words from comment-code mappings," in *Proc. of the 10th Working Conf. on Mining Software Repositories*. IEEE Press, 2013, pp. 377–386.

[24] R. Lotufo, Z. Malik, and K. Czarnecki, "Modelling the 'hurried' bug report reading process to summarize bug reports," *Empirical Software Engineering*, vol. 20, no. 2, pp. 516–548, 2015.

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

[26] M. Allamanis, E. T. Barr, C. Bird, and C. Sutton, "Suggesting accurate method and class names," in *Proc. of the 2015 10th Joint Meeting on Foundations of Softw. Eng.* ACM, 2015, pp. 38–49.

[27] C. Tantithamthavorn, S. McIntosh, A. E. Hassan, and K. Matsumoto, "Automated parameter optimization of classification techniques for defect prediction models," in *Proc. of the 38th Int'l Conf. on Softw. Eng.* IEEE, 2016, pp. 321–332.

[28] R. Mihalcea, C. Corley, and C. Strapparava, "Corpus-based and knowledge-based measures of text semantic similarity," in *National Conf. on Artificial Intelligence and the 18th Innovative App. of Artificial Intelligence Conf.*, 2006, pp. 775–780.

[29] K. W. Church and P. Hanks, "Word association norms, mutual information, and lexicography," *Computational Linguistics*, vol. 16, no. 1, pp. 22–29, 1990.

[30] R. L. Cilibrasi and P. M. Vitanyi, "The Google similarity distance," *IEEE Trans. on Knowledge and Data Eng.*, vol. 19, no. 3, 2007.

[31] K. Lund and C. Burgess, "Producing high-dimensional semantic spaces from lexical co-occurrence," *Behavior Research Methods, Instruments, & Computers*, vol. 28, no. 2, pp. 203–208, 1996.

[32] Y. Tian, D. Lo, and J. Lawall, "Automated construction of a software-specific word similarity database," in *Conf. on the 2014 IEEE Softw. Maintenance, Reengineering and Reverse Eng.*, 2014, pp. 44–53.

[33] C. Chen, Z. Xing, and X. Wang, "Unsupervised software-specific morphological forms inference from informal discussions," in *Proc. of the 39th Int'l Conf. on Softw. Eng.* IEEE Press, 2017, pp. 450–461.

[34] S. Subramanian, L. Inozemtseva, and R. Holmes, "Live API documentation," in *Proc. of the 36th Int'l. Conf. on Softw. Eng.* ACM, 2014, pp. 643–652.

[35] E. M. Voorhees and D. Harman, "Overview of TREC 2001." in *TREC*, 2001.

0098-5589 (c) 2018 IEEE. Personal use is permitted, but republication/redistribution requires IEEE permission. See http://www.ieee.org/publications_standards/publications/rights/index.html for more information.

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.2018.2876006, IEEE Transactions on Software Engineering

[36] E. Yilmaz, M. Verma, R. Mehrotra, E. Kanoulas, B. Carterette, and N. Craswell, "Overview of TREC 2015 tasks track." in *TREC*, 2015.

[37] J. Gracia and E. Mena, "Web-based measure of semantic relatedness," in *Int'l Conf. on Web Information Systems Engineering*. Springer, 2008, pp. 136–150.

[38] E. D. S. Maldonado, E. Shihab, and N. Tsantalis, "Using natural language processing to automatically detect self-admitted technical debt," *IEEE Trans. on Softw. Eng.*, vol. PP, no. 99, pp. 1–1, 2017.

[39] J. Cohen, "A coefficient of agreement for nominal scales," *Educational and Psychological Measurement*, vol. 20, no. 1, pp. 37–46, 1960.

[40] C. McMillan, M. Grechanik, D. Poshyvanyk, Q. Xie, and C. Fu, "Portfolio: finding relevant functions and their usage," in *Proc. of the 33rd Int'l. Conf. on Softw. Eng.* ACM, 2011, pp. 111–120.

[41] E. W. Weisstein, "Bonferroni correction," *Wolfram Research, Inc.*, 2004.

[42] P. F. Brown, V. J. D. Pietra, S. A. D. Pietra, and R. L. Mercer, "The mathematics of statistical machine translation: Parameter estimation," *Computational Linguistics*, vol. 19, no. 2, pp. 263–311, 1993.

[43] K. Papineni, S. Roukos, T. Ward, and W.-J. Zhu, "BLEU: a method for automatic evaluation of machine translation," in *Proc. of the 40th Annual Meeting on Association for Computational Linguistics*. Association for Computational Linguistics, 2002, pp. 311–318.

[44] I. Sutskever, O. Vinyals, and Q. V. Le, "Sequence to sequence learning with neural networks," vol. 4, pp. 3104–3112, 2014.

[45] Apache, "Apache lucene," http://lucene.apache.org/, 2018.

[46] C. Parnin, C. Treude, L. Grammel, and M. A. Storey, "Crowd documentation: Exploring the coverage and the dynamics of API discussions on Stack Overflow," *Georgia Institute of Technology, Tech. Rep*, 2012.

[47] C. Treude and M. P. Robillard, "Augmenting API documentation with insights from Stack Overflow," pp. 392–403, 2016.

[48] C. S. Corley, K. Damevski, and N. A. Kraft, "Exploring the use of deep learning for feature location," in *2015 IEEE Int'l Conf. on Softw. Maintenance and Evolution*. IEEE, 2015, pp. 556–560.

[49] S. Beyer and M. Pinzger, "Synonym suggestion for tags on Stack Overflow," in *IEEE Int'l Conf. on Program Comprehension*, 2015, pp. 94–103.

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

[51] Y. Tian, D. Lo, and J. Lawall, "SEWordSim: software-specific word similarity database," in *Companion Proc. of the 36th Int'l Conf. on Softw. Eng.*, 2014.

[52] S. Wang, D. Lo, and L. Jiang, "Active code search: incorporating user feedback to improve code search relevance," in *ACM/IEEE Int'l Conf. on Automated Softw. Eng.*, 2014, pp. 677–682.

[53] E. Hill, M. Roldanvega, J. A. Fails, and G. Mallet, "NL-based query refinement and contextualized code search results: A user study," in *Softw. Maintenance, Reengineering and Reverse Eng.*, 2014, pp. 34–43.

[54] M. Roldan-Vega, G. Mallet, E. Hill, and J. A. Fails, "CONQUER: A tool for NL-based query refinement and contextualizing code search results," in *IEEE Int'l Conf. on Softw. Maintenance*, 2013, pp. 512–515.

[55] M. Lu, X. Sun, S. Wang, D. Lo, and Y. Duan, "Query expansion via WordNet for effective code search," in *IEEE Int'l Conf. on Software Analysis, Evolution and Reengineering*, 2015, pp. 545–549.

[56] B. A. Campbell and C. Treude, "NLP2Code: Code snippet content assist via natural language tasks," in *Tool Demo of 2017 IEEE Int'l Conf. on Softw. Maintenance and Evolution*, 2017.

[57] I. Keivanloo, J. Rilling, and Y. Zou, "Spotting working code examples," in *Proc. of the 36th Int'l Conf. on Softw. Eng.* ACM, 2014, pp. 664–675.

Xiaochen Li received the BS degree in software engineering from the Dalian University of Technology, China in 2015. He is currently a PhD candidate in Dalian University of Technology. He is a student member of the China Computer Federation (CCF). His current research interests are Mining Software Repositories (MSR) and log analysis in software engineering. He has published several papers, including at some premier conferences in software engineering like ICSE and ICPC. More information about him is available online at http://oscar-lab.org/people/%7excli/.

He Jiang received the PhD degree in computer science from the University of Science and Technology of China, China. He is currently a Professor in Dalian University of Technology, China. He is also a member of the ACM and the CCF (China Computer Federation). He is one of the ten supervisors for the Outstanding Doctoral Dissertation of the CCF in 2014. His current research interests include Search-Based Software Engineering (SBSE) and Mining Software Repositories (MSR). His work has been published at premier venues like ICSE, SANER, and GECCO, as well as in major IEEE transactions like TSE, TKDE, TSMCB, TCYB, and TSC.

Yasutaka Kamei is an associate professor at Kyushu University in Japan. He has been a research fellow of the JSPS (PD) from July 2009 to March 2010. From April 2010 to March 2011, he was a postdoctoral fellow at Queens University in Canada. He received his B.E. degree in Informatics from Kansai University, and the M.E. degree and Ph.D. degree in Information Science from Nara Institute of Science and Technology. His research interests include empirical software engineering, open source software engineering and Mining Software Repositories (MSR). His work has been published at premier venues like ICSE, FSE, ESEM, MSR and ICSM, as well as in major journals like TSE, EMSE, and IST. He will be a program committee co-chair of the 15th International Conference on Mining Software Repositories (MSR 2018). More information about him is available online at http://posl.ait.kyushu-u.ac.jp/%7ekamei/.

Xin Chen was born in 1987. He is currently a faculty at School of Computer Science and Technology, Hangzhou Dianzi University, China. His research interests include software testing and Mining Software Repositories (MSR). He is a member of the China Computer Federation (CCF).

0098-5589 (c) 2018 IEEE. Personal use is permitted, but republication/redistribution requires IEEE permission. See http://www.ieee.org/publications_standards/publications/rights/index.html for more information.

Supplemental Material Bridging Semantic Gaps between Natural Languages and APIs with Word Embedding

Xiaochen Li, He Jiang, *Member, IEEE,* Yasutaka Kamei, *Member, IEEE,* and Xin Chen,

Abstract—This is a supplement material for the paper "Bridging Semantic Gaps between Natural Languages and APIs with Word Embedding". This material includes additional discussions and experiments in different aspects of our approach Word2API. we use Section 1, Section 2 to denote the sections in the main paper and use Section S1, Section S2 to denote the sections in this material.
Part 1. In Section S1, we discuss the selection of the kernel models for Word2API. It is a supplement for Section 3.
Part 2. From Section S2 to S6, we analyze Word2API in relatedness estimation between a word and an API. This part is a supplement for the experiments in Section 5. Specifically, Section S2 compares Word2API with a similar model API2Vec. Then, we discuss the influence of the shuffling times (Section S3), the number of iterations (Section S4), and the tuple length (Section S5) on Word2API. In Section S6, we present the robustness of these experiments by evaluating Word2API over more evaluation metrics. In this part, some additional human judgements for word-API relatedness are conducted.
Part 3. We analyze Word2API in API sequences recommendation. This part provides additional discussions for Section 6. A new strong baseline, namely a deep learning approach DeepAPI, is compared in Section S7. We discuss the ability of Word2API in recommending project-specific APIs in Section S8.
Part 4. We analyze Word2API in API documents linking. This is the task introduced in Section 7. We integrate Word2API into a state-of-the-art approach JBaker for more accurate API documents linking in Section S9.

S1 MODEL SELECTION: CBOW VS. SKIP-GRAM

In the existing studies, two typical models are widely used for word embedding, i.e., CBOW and Skip-gram [1]. In this study, we use CBOW to generate word and API vectors. This section compares the two models, including the efficiency in model training and the effectiveness in performance.

Efficiency. CBOW is more efficient in training than Skip-gram. In this study, we train word embedding with a training set of 138,832,300 word-API tuples. As shown in Table 1, CBOW takes 62 minutes for training. The training speed is 518.91 words per thread·second. The training time is about three times shorter than Skip-gram, which takes 191 minutes for training with a speed of 156.64 words per thread·second. Skip-gram is slower, as it tries to recover every surrounding word with the center word. The model complexity is directly proportional to the number of words in a window [1]. In contrast, CBOW takes the surrounding words as a whole to infer to center word. The window size has fewer influence on its complexity [1]. A faster model is useful in real scenarios [2], especially for parameter optimization in designing a task-specific Word2API model.

Effectiveness. We find the two models yield similar performance in this study. For example, Table 1 compares CBOW and Skip-gram on the task of API documents linking. For this task, MAP and MMR of CBOW are 0.402 and 0.433, which slightly outperform Skip-gram by 0.017 and 0.028 respectively. Although existing studies have compared CBOW and Skip-gram on diverse tasks [1], [3], it is still an open question on which model is more effective.

TABLE 1: Comparison on CBOW and Skip-gram.

Model	Training		Performance	
	Time	Speed	MAP	MMR
CBOW	62 min	518.94 words/thread/sec	0.402	0.433
Skip-gram	191 min	156.64 words/thread/sec	0.385	0.405

Based on above observations, we select the default model CBOW, which achieves similar performance in less time.

S2 COMPARISON OF WORD2API AND API2VEC

In this section, we introduce API2Vec and its differences from Word2API. We also design an experiment to compare the two approaches.

S2.1 Intrinsic Comparison

Tien et al. [4] propose API2Vec to convert APIs into vectors. It is useful to mine API relationships of different programming languages. A typical application of API2Vec is code migration, e.g., migrating APIs from Java to C#.

API2Vec constructs API vectors for different programming languages, e.g., Java and C#, as follows. It first separately trains Java and C# API embedding (vectors) with large-scale Java and C# source code respectively. Then, it

- *X. Li and H. Jiang are with School of Software, Dalian University of Technology, Dalian, China, and Key Laboratory for Ubiquitous Network and Service Software of Liaoning Province. H. Jiang is also an adjunct professor in Beijing Institute of Technology. E-mail: li1989@mail.dlut.edu.cn, jianghe@dlut.edu.cn*
- *Y. Kamei is with the Principles of Software Languages Group (POSL), Kyushu University, Japan. Email: kamei@ait.kyushu-u.ac.jp*
- *X. Chen is with School of Computer Science and Technology, Hangzhou Dianzi University. E-mail: chenxin4391@mail.dlut.edu.cn*

manually labels a set of API mappings between Java and C# that implement the same function, e.g., FileReader#close in Java is the same as StreamReader#Close in C#. With the vectors of the mapping APIs, API2Vec trains a transformation matrix between Java and C# vectors. This matrix can transform unlabeled Java API vectors into the C# vector space, thus the vectors of Java and C# APIs are in the same space. We can use these transformed Java API vectors to calculate the similarity between Java and C# APIs.

Word2API and API2Vec are different in the target and the learning strategy. For the target, Word2API targets at mining relationships between words and APIs instead of APIs and APIs. For the learning strategy, API2Vec is supervised. API2Vec needs to manually label a set of API mappings for training. However, as to our knowledge, no public data set is available to map words with their semantically related APIs. To address this issue, Word2API uses an unsupervised way to analyze word-API relationships.

S2.2 Performance Comparison

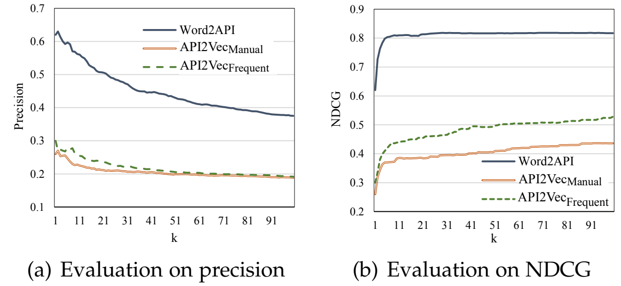
Motivation. In addition to the intrinsic comparison, we experimentally compare API2Vec with Word2API by adapting API2Vec to analyze word-API relationships.

Method. Following the process of API2Vec, we train API2Vec on the word sequences and API sequences with the word-API tuples constructed in Section 3.2. We generate a set of word vectors from the word sequences with the default parameters of the word embedding tool. Similarly, a set of API vectors can be generated according to the API sequences. To transform word vectors to API vectors, we consider two types of word-API mappings to train the transformation matrix, including API2Vec$_{manual}$ and API2Vec$_{freqent}$

API2Vec$_{manual}$ uses manually labeled word-API mappings to calculate the transform matrix. In this paper, we compare Word2API with LSA, PMI, NSD and HAL by recommending APIs to a query word. We manually label the relatedness between 50 query words and the recommended APIs in Section 4.3.3 for evaluation. We use these manually labeled relationships as the training set. We partition the query words into ten folds. Each time, we use 45 words and their related APIs to calculate the transformation matrix, and then transform the remaining 5 words into the API space with the matrix to find their related APIs. On average, the transformation matrix is trained with 3,800 manually labeled word-API mappings.

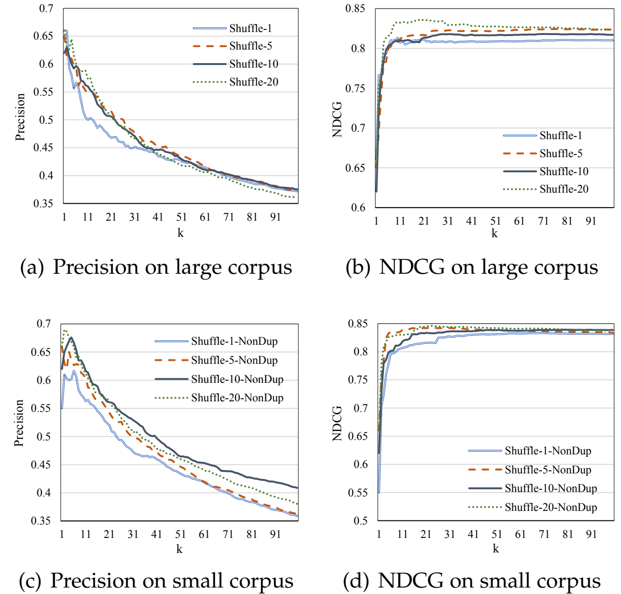
API2Vec$_{freqent}$ uses the frequent 2-itemsets that contain a word and an API as the labeled word-API mappings to calculate the transformation matrix. The detail to mine frequent itemsets is presented in Section 5.3.2. After training, we transform all the 50 query words into the API space with the matrix to find their related APIs. For this method, the training set has 48,961 word-API mappings. We calculate the transformation matrix with Matlab.

Result. As shown in Fig. 1, API2Vec$_{freqent}$ is superior to API2Vec$_{manual}$. The small number of manually labeled word-API mappings may limit the training of API2Vec$_{manual}$. For Word2API, it significantly outperforms the two variants of API2Vec by up to 0.36 in terms of Precision@1 and NDCG@1. We analyze the reason as follows. APIs in different languages are usually one-to-one mappings, i.e., an API in the source language is corresponding to a specific API in the target language. In contrast, the relationship between words and APIs are many-to-many. In the manually labeled training set, each word is considered to be related to 86 APIs on average. Such complex relationship may not be captured by the two-dimensional transformation matrix in API2Vec.



(a) Evaluation on precision (b) Evaluation on NDCG

Fig. 1: Comparison with API2Vec.



(a) Precision on large corpus (b) NDCG on large corpus

(c) Precision on small corpus (d) NDCG on small corpus

Fig. 2: Influence on shuffling times.

Conclusion. In the setting of mining word-API relationships, Word2API can better capture the many-to-many mappings between words and APIs compare to API2Vec.

S3 Influence of Shuffling Times

S3.1 Shuffling on Large Corpus

Motivation. To increase semantically related collocations, Word2API repeats the shuffling step ten times to generate ten shuffled copies of a word-API tuple. This section investigates the influence of the shuffling times on Word2API.

Method. Initially, we collect 13,883,230 word-API tuples from the GitHub corpus. For each word-API tuple, we control the shuffling time from 1 to 20 times, including 1, 5, 10, and 20 times. For example, when the shuffling time is 20, it means we generate 20 shuffled copies of an original

TABLE 2: Shuffling times for API documents linking.

Strategy	MAP	MRR
Shuffle-1	0.368	0.380
Shuffle-5	0.406	0.422
Shuffle-10	0.402	0.433
Shuffle-20	0.416	0.432
Shuffle-1-NonDup	0.354	0.362
Shuffle-5-NonDup	0.393	0.406
Shuffle-10-NonDup	0.402	0.423
Shuffle-20-NonDup	0.410	0.427

word-API tuple. We name this strategy as “Shuffle-20”. It generates 277,664,600 results for training.

Result. The influence of shuffling times on recommending APIs for 50 selected query words is shown in Fig. 2(a) and Fig. 2(b). Clearly, the performance of Shuffle-1 drops from Precision@5 to Precision@30. When we increase the shuffling times, the performance tends to be similar. Similarly, Shuffle-1 also slightly drops in terms of NDCG. However, the differences of different shuffling times are small. The average difference from NDCG@1 to NDCG@100 between Shuffle-1 and Shuffle-20 is 0.018. The small differences between different shuffling times can be also verified on the task of API documents linking (in Table 2). We use this task for re-verification, because the oracle of this task is automatically generated with fewer human biases. Since “Shuffle-20” significantly increases the training time, we shuffle each tuple ten times in this study.

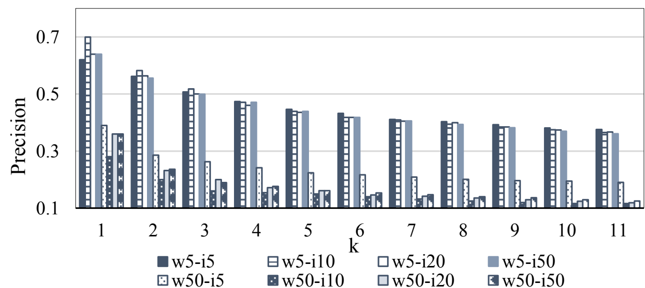
Conclusion. Word2API can be improved by shuffling each word-API tuple multiple times. The performance tends to be stable when the shuffling times vary from 5 to 20.

S3.2 Shuffling on Small Corpus

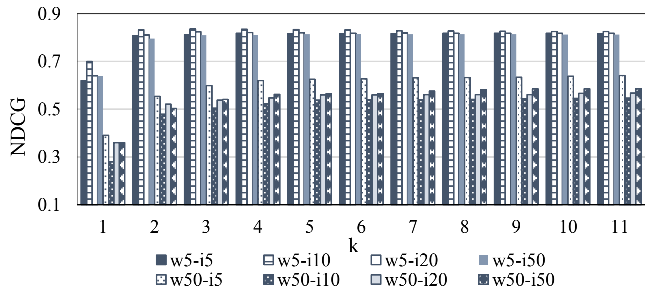
Motivation. As a basic characteristic of GitHub, a project may have many forks or third-party source code [5], leading to many duplicate code snippets. To better analyze the influence of shuffling times, in this subsection, we generate a small corpus by removing the duplications in the large corpus and analyze the influence of shuffling times on the small corpus.

Method. We calculate the MD5 value of each word-API tuple in the large corpus. We remove the duplicate copies of word-API tuples that have the same MD5 value. In this way, we obtain 5,488,201 non-duplicate word-API tuples, i.e., the duplicate rate is 0.605. Then, we train Word2API on the non-duplicate word-API tuples by shuffling each tuple 1, 5, 10, 20 times, denoted as Shuffle-1-NonDup, Shuffle-5-NonDup, Shuffle-10-NonDup and Shuffle-20-Nondup respectively.

Result. As shown in Fig. 2(c) and Fig. 2(d), when increasing the shuffling times, the performance of Word2API slightly improves, and then reaches a ceiling. When we apply the vectors generated by these variants on the task of API documents linking, we can observe similar trends (in Table 2). In addition, by comparing the performance of Word2API on the large and small corpora in Table 2, we find that the absence of code duplication negatively affects the Word2API performance on API documents linking.



(a) Precision on the number of iterations



(b) NDCG on the number of iterations

Fig. 3: Influence on the number of iterations.

TABLE 3: The number of iterations for API documents linking.

Strategy	MAP	MRR
Word2API-w5-i5	0.402	0.433
Word2API-w5-i10	0.413	0.430
Word2API-w5-i20	0.405	0.420
Word2API-w5-i50	0.412	0.427
Word2API-w50-i5	0.205	0.214
Word2API-w50-i10	0.205	0.211
Word2API-w50-i20	0.194	0.200
Word2API-w50-i50	0.205	0.209

Conclusion. As a machine learning approach, the corpus size influences Word2API in learning word-API relationships. When training Word2API on a small corpus (5,488,201 non-duplicate word-API tuples), the performance of Word2API for solving the API documents linking problem slightly drops.

S4 Influence on the Number of Iterations

Motivation. This section investigates how the number of iterations influences Word2API.

Method. By default, the number of iterations of Word2API is 5. We increase the number of iterations (denoted as i) by 5, 10, 20, 50 and observe the performance of Word2API on recommending APIs according to query words. In this experiment, the default window size (denoted as w) is 5. Hence, the algorithms include Word2API-w5-i5, Word2API-w5-i10, Word2API-w5-i20, and Word2API-w5-i50.

Besides, we also set the window size to 50, since the performance of Word2API sharply drops when the window size increases from 5 to 50 (see Section 5.2.1). We observe the influence of the number of iterations on this larger window size.

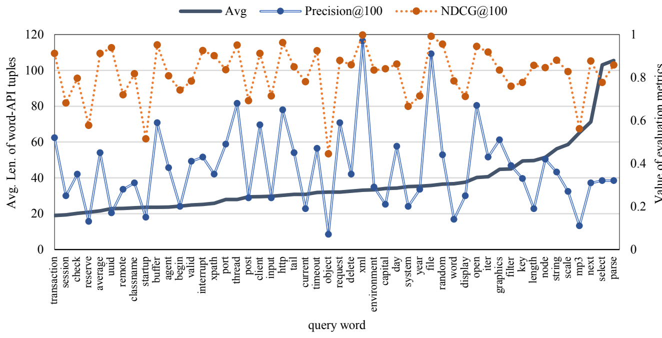


Fig. 4: The average tuple length and the performance.

Result. As shown in Fig. 3(a) and Fig. 3(b), the number of iterations has little influence on Word2API when the window size is 5. If we average the differences between Word2API-w5-i5 and Word2API-w5-i50 for the ranking list from 1 to 100, the average difference between i = 5 and i = 50 is 0.006 for precision and 0.003 for NDCG. Conversely, when the window size is set to 50, increasing the number of iterations decreases the performance of Word2API. However, such differences do not affect the overall applicability of Word2API for solving software engineering tasks. When these variants of Word2API are applied to API documents linking, the performance of Word2API is stable as the number of iterations is tuned, as shown in Table 3.

Conclusion. WordAPI is robust to the number of iterations for software engineering tasks.

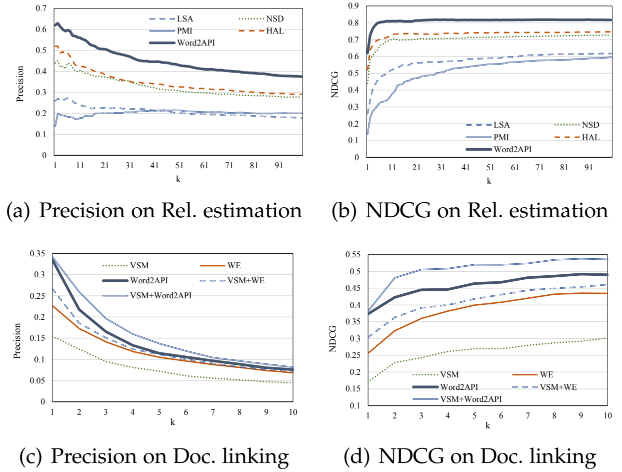
S5 Influence on the Tuple Length

Motivation. This section investigates how the length of word-API tuples influences Word2API.

Method. Given a query word in the 50 selected ones, we collect all the word-API tuples containing this word. We calculate the average length (number of terms) of the collected word-API tuples, as well as the performance of Word2API on recommending related APIs for this word. Then, we observe the correlation between the two variables.

Result. The results are presented in Fig. 4. The x-axis is the query word. We rank the query words according to the average tuple length containing each word. The left y-axis is the value of the average length of tuples. The right y-axis shows the values of Precision@100 and NDCG@100 with respect to each query word. We find these query words are trained on tuples with diverse lengths. The average length of tuples containing the word "transaction" is 18.98. In contrast, the word "parse" is trained by many long tuples. The average length is 105.52. Despite the diverse lengths, we could not observe a correlation between the tuple length and the performance. The Spearman correlation coefficient is -0.022 between the average tuple length and Precision@100 and 0.026 between the average tuple length and NDCG@100.

Conclusion. The length of tuples may not be a core factor to influence the performance of Word2API.



(a) Precision on Rel. estimation (b) NDCG on Rel. estimation

(c) Precision on Doc. linking (d) NDCG on Doc. linking

Fig. 5: Precision and NDCG for API relatedness estimation and API documents linking.

TABLE 4: MAP and MRR for API relatedness estimation and API documents linking.

Task	Algorithms	MAP	MRR
API Rel. estimation	LSA	0.210	0.242
	PMI	0.259	0.396
	NSD	0.293	0.491
	HAL	0.301	0.488
	Word2API	0.362	0.528
API Doc. linking	VSM	0.232	0.259
	WE [6]	0.313	0.354
	Word2API	0.402	0.433
	VSM+WE [6]	0.340	0.380
	VSM+Word2API	0.436	0.469

S6 Evaluation over More Metrics

Motivation. To show the robustness of Word2API, we use precision, NDCG, MAP, and MRR to conduct a thorough evaluation on the tasks of word-API relatedness estimation (Section 5) and API documents linking (Section 7).

Method. For word-API relatedness estimation, we select 50 query words to compare Word2API against the baselines, including LSI, PMI, NSD, and HAL. These algorithms are evaluated by recommending 100 APIs corresponding to a query word. For API documents linking, we compare Word2API against VSM and WE. The algorithms are evaluated by recommending 10 API documents to a question in Stack Overflow. We show the performance of both the two tasks on precision, NDCG, MAP, and MRR.

Result. Fig. 5(a) and Fig. 5(b) are the averaged precision and NDCG for different algorithms on API relatedness estimation. We show MAP and MRR for this task in Table 4. Clearly, Word2API outperforms the baselines in terms of all the evaluation metrics. These metrics evaluate Word2API in different aspects. Precision and MAP count the percentage of related APIs in a ranking list. MRR focuses on the position of the related APIs and NDCG compares the position of the related APIs with the unrelated ones.

We observe similar results for the task of API documents linking. We present the performance of different algorithms for API documents linking in Fig. 5(c), Fig. 5(d) and Table 4.

In this task, Word2API is superior to VSM and WE over all the evaluation metrics.

Conclusion. The effectiveness of Word2API in capturing the semantic relatedness can be verified over diverse evaluation metrics.

S7 Comparison with Deep API Learning

In this section, we compare Word2API with the state-of-the-art algorithm for API sequences recommendation and discuss the differences between the two algorithms to justify the application scenario of Word2API.

S7.1 Quantified Comparison

Motivation. Word2API is a component for sematic estimation. We integrate Word2API into a Lucene$_{API}$ (Section 6.4.2) based search framework to show how Word2API works for practical API recommendation. This method is denoted as Word2API$_{Search}$. We compare Word2API$_{Search}$ with DeepAPI [7], an attention-based RNN Encoder-Decoder algorithm for API sequences recommendation.

Method. *DeepAPI* learns word-API relationships from word-API tuples constructed from the GitHub corpus. For a word-API tuple, DeepAPI takes the words in the word sequence as input. It encodes and decodes these words with an RNN network and outputs a set of vectors representing the related API sequences regarding these words. To train the RNN network, DeepAPI optimizes the parameters of RNN by minimizing the differences between the output API sequence and the actual API sequence in this word-API tuple. Finally, DeepAPI achieves a set of optimized parameters. In evaluation, DeepAPI encodes and decodes the vector of a user query with the optimized RNN and directly generates API sequences for the query. Gu et al. [7] published an on-line demo of DeepAPI[1] for evaluation.

Word2API$_{Search}$ integrates Word2API into the widely used search engine Lucene for practical API sequences recommendation. Word2API$_{Search}$ first expands a user query into a combined query with both words and related APIs. It uses this combined query to search candidate API sequences from the word-API tuples. Then, Word2API$_{Search}$ re-ranks the candidate API sequences by both sematic similarity and text similarity, and recommends the top ranked API sequences.

Specifically, we use Word2API to calculate the similarity between a user query and each API$_i$ in Java SE APIs, denoted as sim_{API_i}. We combine a user query and the top-10 APIs with the largest sim_{API_i} to form a combined query q_{com}. The top-10 APIs are selected as suggested by the previous study [8]. We search q_{com} with Lucene to get top 1,000 candidate API sequences in word-API tuples. This step uses the text information of q_{com}, i.e., Term Frequency and Inverted Document Frequency (IDF), to filter low-quality and noisy API sequences. The words in q_{com} are used to match the words split from the API sequences. The APIs in q_{com} are used to directly match the APIs in API sequences.

Then, we re-rank the candidate API sequences with the assistance of the sematic information, i.e., sim_{API_i}. This process is inspired by Lv. et al. [8]. We do not directly use their model, as the original model has several parameters, which needs to be carefully optimized on different tasks.

1. https://guxd.github.io/deepapi/. Last check June, 2018.

TABLE 5: Performance of DeepAPI and Word2API$_{Search}$

ID	DeepAPI [7]			Lucene$_{API}$			Word2API$_{Search}$		
	FR	P@5	P@10	FR	P@5	P@10	FR	P@5	P@10
Q1	2	0.4	0.9	NF	0	0	NF	0	0
Q2	1	1	1	NF	0	0	2	0.8	0.9
Q3	1	1	1	1	1	1	1	1	1
Q4	10	0.1	0.1	1	1	1	1	1	1
Q5	1	1	0.8	NF	0	0	1	1	1
Q6	1	1	1	NF	0	0	1	1	1
Q7	1	1	1	1	1	1	1	1	1
Q8	1	1	0.8	1	1	1	1	1	1
Q9	3	0.4	0.5	NF	0	0	1	0.6	0.4
Q10	1	0.8	0.9	NF	0	0	1	1	0.7
Q11	1	1	1	NF	0	0	1	0.6	0.8
Q12	1	1	0.7	1	0.6	0.6	1	1	1
Q13	1	1	1	5	0.2	0.6	1	1	1
Q14	1	0.8	0.6	1	0.8	0.9	1	1	1
Q15	1	1	0.9	NF	0	0	1	1	1
Q16	3	0.4	0.2	NF	0	0	1	1	0.6
Q17	2	0.2	0.1	NF	0	0	1	1	1
Q18	1	1	1	NF	0	0	1	1	1
Q19	1	1	1	NF	0	0	1	1	1
Q20	2	0.6	0.7	NF	0	0	1	1	0.7
Q21	1	0.6	0.8	1	1	0.7	1	0.6	0.6
Q22	1	1	1	NF	0	0	1	1	1
Q23	1	1	0.8	9	0	0.2	7	0	0.2
Q24	3	0.6	0.7	4	0.4	0.2	1	1	1
Q25	1	1	0.8	7	0	0.1	1	0.4	0.4
Q26	1	0.8	0.8	1	1	1	1	1	1
Q27	1	1	0.9	1	1	1	1	1	1
Q28	1	0.8	0.6	5	0.2	0.1	1	1	1
Q29	1	0.6	0.8	6	0	0.2	1	1	1
Q30	1	1	0.9	4	0.4	0.7	NF	0	0
Avg.	1.6	0.8	0.78	6.767	0.320	0.343	1.9	0.833	0.81
p	1.0	0.65	0.453	<0.01	<0.01	<0.01	*	*	*

We simplify their model as follows. This model ranks a candidate API sequence by the sum of its sematic similarity and text similarity to the query [8]. The sematic similarity is the sum of sim_{API_i} of all the APIs that appear in both the combined query q_{com} and the candidate API sequence seq.

$$sim_{semantic} = \sum_{i=1}^{k} sim_{API_i},\ API_i \text{ appears in } q_{com} \text{ and } seq. \quad (1)$$

For the text similarity, the weight of word$_i$ in q_{com} is defined as:

$$sim_{word_i} = \log(IDF_{word_i}) / \sum_{j=1}^{n} \log(IDF_{word_j}), \quad (2)$$

where n is the number of words in q_{com} and IDF_{word_i} is the IDF of $word_i$. Similar to $sim_{semantic}$, the text similarity is the sum of sim_{word_i} of all words that appear in both q_{com} and seq. We split seq into words according to their camel style.

$$sim_{text} = \sum_{i=1}^{k} sim_{word_i},\ word_i \text{ appears in } q_{com} \text{ and } seq. \quad (3)$$

The final similarity between the user query q and seq is:

$$sim(q, seq) = \frac{(sim_{semantic} + sim_{text}) * Num_{matched}}{Len_{seq}}, \quad (4)$$

where $Num_{matched}$ is the number of matched terms (APIs and words) in seq and Len_{seq} is the length of seq. $Num_{matched}$ is used to improve the influence of word-API sequences that can match more terms, as the previous study [8] assumes APIs that are retrieved by multiple terms more important. Len_{seq} is used to lessen the influence of long API sequences, which can always match more terms.

Result. Table 5 presents the performance of DeepAPI, Lucene$_{API}$, and Word2API$_{Search}$ over the human written

queries. Lucene_{API} is the algorithm evaluated in Section 6.4.2. Word2API_{Search} improves the performance of Lucene_{API} by 0.513 and 0.467 in terms of P@5 and P@10 respectively. Hence, it is promising to integrate the semantic information analyzed by Word2API into a general-purpose search engine. When comparing Word2API_{Search} with DeepAPI, we could not observe statistical differences between the two algorithms. They both achieve the state-of-the-art results for API sequences recommendation over the real-world queries. We did not evaluate these algorithms with the 10,000 automatically constructed queries, as the DeepAPI demo was down when we sent our constructed queries.

Conclusion. Word2API_{Search} performs similar with the state-of-the-art algorithm DeepAPI.

S7.2 Qualitative Comparison

Motivation. Since Word2API_{Search} and DeepAPI perform similar over the real-world queries, we conduct a qualitative comparison of the two algorithms to provide some insights on utilizing Word2API_{Search}.

Method. We analyze the failure cases of Word2API_{Search} in recommending APIs, and then discuss the application scenario of Word2API_{Search}.

Result. We analyze Word2API_{Search} in three aspects.

First, Word2API_{Search} takes a query as bag-of-words. It misses the knowledge of the order of words in a query. Hence, Word2API_{Search} fails to distinguish the query Q1 "convert int to string" from Q2 "convert string to int". This is a common problem of bag-of-words based models [8].

Second, Word2API_{Search} may not well handle some queries with multiple requirements. For example, the query Q30 "play the audio clip at the specified absolute URL" has two requirements, including "play the audio clip" and "at the specified absolute URL". When searching this query, Word2API_{Search} lowers down the weight (IDF) of the second requirement, as "URL" is a common word to describe "java.net" packages. As a result, Word2API_{Search} only recommends APIs related to "play the (local) audio clip" instead of the "on-line" ones.

Third, as a retrieval task, Word2API_{Search} may suffer from poor-quality queries, that are far from the human intention.

Despite the above shortcomings, Word2API_{Search} is still competitive to used. We discuss the potential advantages of Word2API_{Search} by comparing Word2API_{Search} with DeepAPI.

First, DeepAPI is a deep neural network based method. The reasons for generating an API sequence is usually opaque to developers [9]. In contrast, Word2API_{Search} recommends API sequences by ranking word-API tuples. Most parts of Word2API_{Search} are explainable. Developers could understand the recommendation results and optimize the model in different scenarios more easily.

Second, DeepAPI generates API sequences by network parameters. On the one hand, the generative model DeepAPI can infer new API sequences after training on historical API sequences. This is useful for developers seeking to learn the new usages of APIs. In this respect, DeepAPI is superior to Word2API, which only recommends existing historical API sequences. On the other hand, after manually examining the generated API sequences by DeepAPI, we find that some API sequences may not be valid, which may be a burden in understanding and debugging these sequences. In this respect, Word2API_{Search} can retrieve valid and real-world API sequences. These sequences can be directly linked to the source code for better understanding.

Conclusion. Compared to DeepAPI, Word2API is useful in finding real-world API sequences. The recommendation results are more explainable.

S8 Learning on Project-Specific APIs

Motivation. This study trains Word2API on Java SE APIs. Since searching for Java SE APIs has been well studies by general-purpose search engines, this section investigates Word2API on learning project-specific words and APIs.

Method. We take the core Lucene APIs as a representative example of project-specific APIs. On the one hand, Lucene is widely known to developers. Recommending Lucene APIs is helpful to set up a general-purpose search engine. On the other hand, compared to Java SE APIs, core Lucene APIs are not used in all the Java projects. Searching for Lucene APIs is more similar to a project-specific search.

In the experiment, we collect the code snippets containing Lucene APIs from the GitHub corpus. Similar to the process of constructing Java SE word-API tuples, we construct word-API tuples for core Lucene APIs. In this process, we collect 94,571 word-API tuples. We generate a training set by creating ten copies of each word-API tuple with the shuffling strategy. After running Word2API on the training set, 3,088 word vectors and 8,279 API vectors are generated eventually.

In the evaluation, we first evaluate Word2API with 30 human written queries listed in the first three columns of Table 6. The typical APIs for each query are listed in the forth column. The first five queries are the general steps to deploy a Lucene search engine in the Lucene tutorial[2]. The remaining queries are selected from the title of top voted questions in Stack Overflow with the tag "Lucene". We select queries according to the following criteria [10]: (1) The question is a programming task that can be implemented with core Lucene APIs. (2) The answer to the question contains Lucene APIs. (3) The title of the question is not the same with the already selected queries. Then, we expand the selected queries into API vectors and search word-API tuples based on the naive framework presented in Section 6.2.3 (Word2API_{Exp}) to highlight the affect of Word2API. The top-10 results are evaluated by FR, Precision@5, and Precision@10.

Second, we randomly select 1,000 word-API tuples from all the 94,571 word-API tuples. We only select 1,000 word-API tuples, due to the small number of entire Lucene related tuples. We take the word sequences in the word-API tuples as queries to search API sequences in the remaining 93,571 word-API tuples. The recommended API sequences are evaluated based on the BLEU score.

We compare Word2API_{Exp} with $\text{Lucene}_{API+Comment}$ proposed in Section 6.4.2. $\text{Lucene}_{API+Comment}$ in this section only searches Lucene word-API tuples. It matches the queries

2. https://www.tutorialspoint.com/lucene/lucene_overview.htm

TABLE 6: Performance on project-specific search over 30 human written queries. P is short for precision

ID	Query (How to/Is there a way for)	Question ID	Typical APIs	Lucene$_{API+Comment}$ FR	P@5	P@10	Word2API$_{Exp}$ FR	P@5	P@10
L1	analyze the document	tutorial	StandardAnalyzer#new, Analyzer#tokenstream	1	1	1	1	0.8	0.8
L2	indexing the document	tutorial	IndexWriterConfig#new, IndexWriter#new, IndexWriter#addDocument	1	0.8	0.7	1	1	1
L3	build query	tutorial	BooleanClause#getQuery, QueryParser#parse	1	0.8	0.8	1	0.4	0.6
L4	search query	tutorial	IndexSearcher#search	1	1	0.8	1	0.8	0.7
L5	render results	tutorial	Explanation#getSummary, Explanation#getDetails	5	0.2	0.4	1	1	0.8
L6	get a token from a lucene TokenStream	2638200	TokenStream#incrementToken, TermAttribute#term	3	0.4	0.5	1	1	1
L7	keep the whole index in RAM	1293368	RAMDirectory#new	NF	0	0	NF	0	0
L8	stem English words with lucene	5391840	EnglishAnalyzer#new, PorterStemmer#stem	3	0.4	0.6	4	0.2	0.5
L9	ignore the special characters	263081	QueryParser#escape	3	0.2	0.1	4	0.4	0.2
L10	incorporate multiple fields in QueryParser	468405	TermQuery#new, BooleanQuery#add, MultiFieldQueryParser#new	1	0.4	0.4	1	1	1
L11	tokenize a string	6334692	Analyzer#tokenStream	1	0.8	0.9	NF	0	0
L12	(use) different analyzers for each field	2843124	PerFieldAnalyzerWrapper#new	NF	0	0	NF	0	0
L13	load default list of stopwords	17527741	StanardAnalyzer#loadStopwordSet	5	0.2	0.4	1	0.8	0.5
L14	sort lucene results by field value	497609	Search#sort, Sort#getSort	2	0.4	0.5	1	0.8	0.5
L15	extract tf-idf vector in lucene	9189179	IndexReader#docFreq, IndexReader#getTermVector, TFIDFSimilarity#idf	3	0.4	0.4	2	0.8	0.9
L16	backup lucene index	5897784	FSDirectory#copy	3	0.2	0.1	NF	0	0
L17	find all lucene documents having a certain field	3710089	QueryParser#SetAllowLeadingWildcard	NF	0	0	NF	0	0
L18	(calculate) precision/recall in lucene	7170854	ConfusionMatrixGenerator#getPrecision, ConfusionMatrixGenerator#getRecall	1	0.8	0.5	1	0.8	0.4
L19	search across all the fields	15170097	TermQuery#new, BooleanQuery#add, MultiFieldQueryParser#new	NF	0	0	5	0.2	0.4
L20	multi-thread with lucene	9317981	MultiReader#new, MultiSearcherThread#start	3	0.4	0.3	1	0.4	0.2
L21	get all terms for a lucene field in	15290980	Fields#terms, Term#text	7	0	0.1	1	1	1
L22	update a lucene index	476231	Document#add, IndexWriter#addDocument	2	0.6	0.6	2	0.8	0.9
L23	adding tokens to a TokenStream	17476674	TokenStream#incrementToken, PositionIncrementAttribute#setPositionIncrement	1	1	0.8	1	0.8	0.8
L24	finding the num of documents in a lucene index	442463	IndexReader#numDocs	1	0.8	0.9	1	0.8	0.9
L25	make lucene be case-insensitive	5512803	StringUtil#startsWithIgnoreCase, LowerCaseFilter#new	3	0.4	0.4	2	0.4	0.4
L26	boost factor (of) MultiFieldQueryParser	551724	MultiFieldQueryParser#new, Query#setBoost	3	0.2	0.2	1	0.4	0.3
L27	list unique terms from a specific field	654155	Term#iterator, TermsEnum#next	8	0	0.1	1	1	0.8
L28	index token bigrams in lucene	8910008	NGramTokenizer#new	NF	0	0	NF	0	0
L29	delete or update a doc	2634873	IndexWriter#update, IndexReader#removeDocument	3	0.4	0.5	1	0.8	0.7
L30	query lucene with like operator	3307890	WildcardQuery#new, PrefixQuery#new	NF	0	0	2	0.4	0.4
Avg.				4.367	0.393	0.4	3.467	0.56	0.523
p				0.067	0.029	0.041	*	*	*

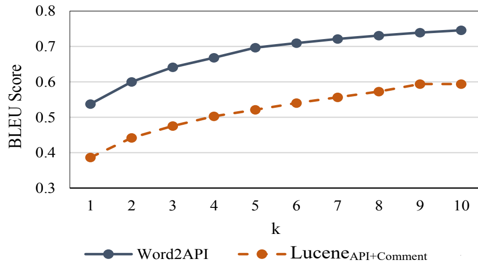


Fig. 6: BLEU score on project-specific search.

with the words in the word sequence and API sequence of each word-API tuple.

Result. Table 6 shows the results on human written queries. For FR, the average position of the first related API sequence recommended by Word2API$_{Exp}$ ranks 0.9 higher than Lucene$_{API+Comment}$. For precision, Word2API$_{Exp}$ outperforms Lucene$_{API+Comment}$ by 0.16 and 0.123 in terms of Precision@5 and Precision@10 respectively. The results on precision pass the Wilconox signed rank test with p-values < 0.05. Similarly, we can also observe a significant improvement in Fig. 6 in terms of the BLEU score over the 1,000 automatically constructed queries. Hence, Word2API$_{Exp}$ outperforms Lucene$_{API+Comment}$ in recommending project-specific APIs over precision and the BLEU score.

Despite the promising results, we analyze the failure cases of Word2API$_{Exp}$ to provide some insights in using Word2API$_{Exp}$. The first failure reason is the small size of vocabulary in the training set. Word2API generates 3,088 word vectors. We find some words in the query never occur in the training set. For example, for the query L11 "tokenize a string", Word2API cannot generate a vector for "tokenize", leading to a failure result. One direction to solve this problem is to infer the software-specific morphological forms of the non-existence words [11], e.g., "token" and "tokenize" come from the same root. We may use the vector of "token" to calculate similarity. Another direction is to combine Lucene$_{API+Comment}$ with Word2API, as Lucene$_{API+Comment}$ finds the right APIs for this query.

The second failure reason is the lack of the diversity of word-API usages. The training set is 100 times smaller than the Java SE training set. Some usages between words and APIs may not exist in the method comments and API calls. For example, we could not observe obvious usages of the word "RAM" to describe "RAMDirectory#new" related APIs (query L7) in the word-API tuples. Although as discussed in Section S3, the shuffling strategy improves the ability of Word2API in learning existing word-API tuples, the non-existence word-API usages may lead to a failure.

Conclusion. Word2API can learn word-API relationships for project-specific APIs. A searching framework with the Word2API-generated queries can provide more precise results than a general-purpose search engine.

S9 API Documents Linking with JBaker

Motivation. Word2API is useful for API documents linking, e.g. linking the questions in Stack Overflow to their related API documents. This section compares Word2API with JBaker on this task, one of the state-of-the-art algorithms of linking on-line resources (e.g., Stack Overflow questions) to API documents. JBaker represents a set of algorithms that trace the exact type (the fully qualified name) of ambiguous

TABLE 7: Performance of JBaker and baselines.

#Exp	Algorithms	MAP	MRR
Group 1	JBaker [12]	0.337	0.344
	JBaker-code	0.448	0.458
Group 2	VSM	0.195	0.195
	WE [6]	0.190	0.187
	Word2API	0.338	0.350
Group 3	JBaker+Word2API	0.501	0.514
	$\text{Google}_{\textit{Specification}}$	0.501	0.509

APIs in code snippets. For example, JBaker can deduce whether the ambiguous API "Data#getHours" in a code snippet refers to "java.util.Data" or "java.sql.Data". Since each API document is usually illustrating an unique API type, JBaker is able to link every ambiguous API in the code snippet to its related API documents.

Method. We use JBaker for API documents linking. For a question in Stack Overflow, we extract the code snippet in the question. We input the code snippet to JBaker for identifying the exact API type of every ambiguous API in the snippet. JBaker analyzes ambiguous APIs based on an oracle. The oracle is a database containing a large number of API sequences used in practice. When JBaker encounters an ambiguous API, it matches the ambiguous API with the API sequences in the oracle to deduce its possible API types. JBaker assumes that APIs in the same code snippet usually belong to the same API type. Hence, it can find the exact type of an ambiguous API by identifying the common API types of all ambiguous APIs. Based on the deduced API type, we link ambiguous APIs to API documents. If JBaker cannot find the exact type of an ambiguous API, it recommends more than one results. Thus, we link this ambiguous API to more than one API document. In this study, we use the API sequences in the word-API tuples as the oracle. We reproduce JBaker by ourselves.

After linking every ambiguous API with API documents, we rank these API documents for the task of API documents linking. We define the score of an API document to a question as the score of all the APIs in the question that are linked to this API document.

$$score_{doc} = \sum_{i=1}^{n} score_{doc_{API_i}}, \tag{5}$$

where n is the number of APIs that are linked to this API document by JBaker. Since JBaker may link an API to more than one API document, the score of an API is defined as:

$$score_{doc_{API_i}} = 1/k_i, \tag{6}$$

where k_i is the number of API documents that JBaker links API_i to. Based on $score_{doc}$, we recommend API documents for a question in Stack Overflow.

Result. As described in Section 7.3.2, we collect 278 questions from Stack Overflow as a testing set for evaluation. Table 7 is the performance of the algorithms.

For *the first group of experiments*, we evaluate JBaker on the 278 questions. The performance of JBaker is 0.337 and 0.344 in terms of MAP and MRR respectively. Recalling that Word2API achieves MAP of 0.402 and MRR of 0.433 on the same testing set, Word2API outperforms JBaker over the 278 questions. We reason the JBaker's performance as follows. On the one hand, despite JBaker can correctly link APIs in code snippets to API documents, these API documents may not be the correct ones to solve the problems, as the submitters may already read these API documents before submitting the question. On the other hand, not all the questions in Stack Overflow contains code snippets. As a statistic of the 278 questions, 70 (25.2%) of them have no code snippets. JBaker may recommend nothing for these questions. If we remove these 70 questions, the performance of JBaker-code on the remaining 208 questions are significantly improved as shown in the 2nd line of Table 7.

However, we think the removed 70 questions are more difficult to analyze. Since these questions only contain natural language words, the gaps between words in questions and APIs in API documents are more prominent. For *the second group of experiments*, we run the algorithms in Section 7.2 on the 70 questions, including VSM, WE, and Word2API. The performance of all the algorithms drops, even though Word2API still outperforms the others by 0.143 to 0163 over distinct metrics. Hence, Word2API can better bridge the sematic gaps than the baselines on some "hard" instances.

Although JBaker may have difficulty in analyzing questions without code snippets, JBaker is useful to analyze the API-API relationship between code snippets and API documents. For *the third group of experiments*, we combine the word-API relationship analyzed by Word2API and the API-API relationship analyzed by JBaker for more precise API documents linking. For a question, we assign two scores to each API document. The scores are calculated by Word2API and JBaker. All the API documents are ranked according to the sum of the two scores (Word2API+JBaker). If a question has no code snippets, JBaker assigns zero to all the API documents. In Table 7, both MAP and MRR of Word2API+JBaker over the 278 questions are significantly improved, i.e., 0.501 for MAP and 0.514 for MRR.

In addition, we compare Word2API+JBaker with Google, a state-of-the-art search engine. We take the 278 questions as queries and manually search Java API documents with Google by rewriting a query as 'query site:https://docs.oracle.com/javase/8/docs/api/'. This method is denoted as $\text{Google}_{\textit{Specification}}$. We find Google provides a strong baseline for information retrieval tasks in software engineering. For API documents linking, the results of $\text{Google}_{\textit{Specification}}$ and Word2API+JBaker are quite close. According to classical information retrieval textbooks [13], a mature search engine may leverage many state-of-the-art techniques to optimize the search results, such as page rank, topic model, query expansion, and query feedback. Hence, the word-API and API-API knowledge captured by Word2API+JBaker is competitive as a combination of many retrieval techniques in analyzing APIs.

Conclusion. Word2API outperforms the baselines over different types of questions. The word-API relationship analyzed by Word2API is valuable to improve the algorithms for API documents linking.

REFERENCES

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

[2] H. Jiang, X. Li, Z. Yang, and J. Xuan, “What causes my test alarm? automatic cause analysis for test alarms in system and integration testing,” in *Proc. of the 39th Int'l. Conf. on Softw. Eng.* IEEE, 2017, pp. 712–723.

[3] F. Asr, J. Willits, and M. Jones, “Comparing predictive and co-occurrence based models of lexical semantics trained on child-directed speech,” in *Proc. of CogSci*, 2016.

[4] T. D. Nguyen, A. T. Nguyen, H. D. Phan, and T. N. Nguyen, “Exploring API embedding for API usages and applications,” in *Proc. of the 39th Int'l. Conf. on Softw. Eng.* IEEE Press, 2017, pp. 438–449.

[5] E. Kalliamvakou, G. Gousios, K. Blincoe, L. Singer, D. M. German, and D. Damian, “An in-depth study of the promises and perils of mining github,” *Empirical Software Engineering*, vol. 21, no. 5, pp. 2035–2071, 2016.

[6] 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 *Proc. of the 38th Int'l Conf. on Softw. Eng.* ACM, 2016, pp. 404–415.

[7] X. Gu, H. Zhang, D. Zhang, and S. Kim, “Deep API learning,” in *Proc. of the 2016 24th ACM SIGSOFT Int'l Symposium on Foundations of Softw. Eng.* ACM, 2016, pp. 631–642.

[8] F. Lv, H. Zhang, J.-g. Lou, S. Wang, D. Zhang, and J. Zhao, “Codehow: Effective code search based on API understanding and extended boolean model,” in *30th IEEE/ACM Int'l Conf. on Automated Softw. Eng.* IEEE, 2015, pp. 260–270.

[9] M. Kahng, P. Y. Andrews, A. Kalro, and D. H. P. Chau, “Activis: Visual exploration of industry-scale deep neural network models,” *IEEE Trans. on Visualization and Computer Graphics*, vol. 24, no. 1, pp. 88–97, 2018.

[10] X. Gu, H. Zhang, and S. Kim, “Deep code search,” in *Proc. of the 40th Int'l. Conf. on Softw. Eng.* ACM, 2018, pp. 933–944.

[11] C. Chen, Z. Xing, and X. Wang, “Unsupervised software-specific morphological forms inference from informal discussions,” in *Proc. of the 39th Int'l Conf. on Softw. Eng.* IEEE Press, 2017, pp. 450–461.

[12] S. Subramanian, L. Inozemtseva, and R. Holmes, “Live API documentation,” in *Proc. of the 36th Int'l. Conf. on Softw. Eng.* ACM, 2014, pp. 643–652.

[13] H. Schütze, C. D. Manning, and P. Raghavan, *Introduction to information retrieval*. Cambridge University Press, 2008, vol. 39.