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

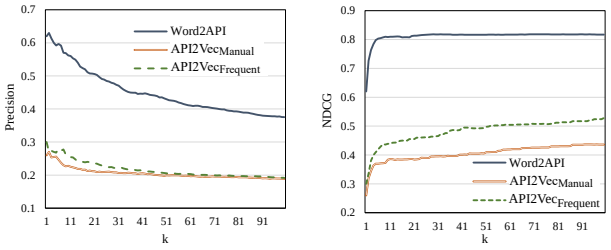
Motivation. In addition to the intrinsic comparison, we experimentally compare API2Vec with Word2API by adapting API2Vec to analyze word-API relationships.

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.

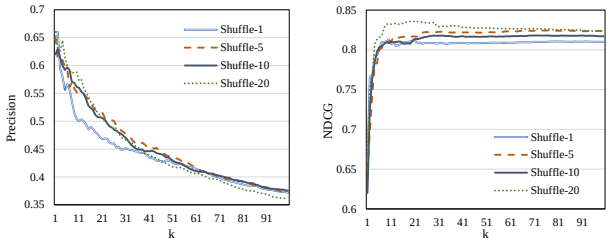
API2Vec$_{freqent}$ uses the frequent 2-itemsets that contain a word and an API as the labeled word-API mappings to calculate the transformation matrix. The detail to mine frequent itemsets is presented in Section 5.3.2. After training, we transform all the 50 query words into the API space with the 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.

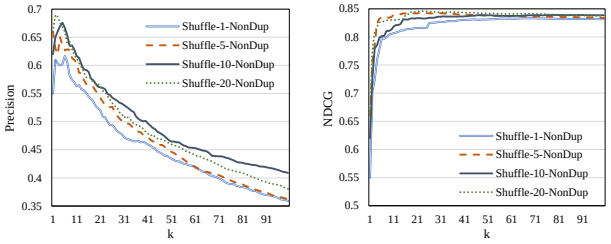


(a) Evaluation on precision (b) Evaluation on NDCG

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.

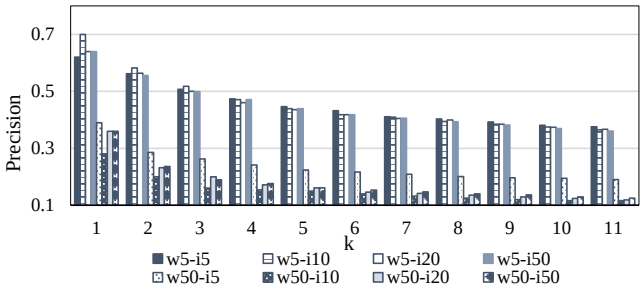
Conclusion. Word2API can be improved by shuffling each word-API tuple multiple times. The performance tends to be stable when the shuffling times vary from 5 to 20.

S3.2 Shuffling on Small Corpus

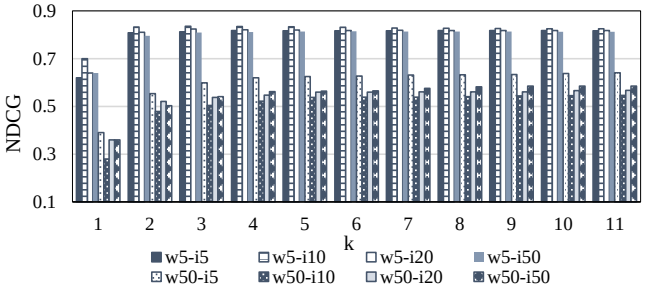
Motivation. As a basic characteristic of GitHub, a project may have many forks or third-party source code [5], leading to many duplicate code snippets. To better analyze the influence of shuffling times, in this subsection, we generate a small corpus by removing the duplications in the large corpus and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

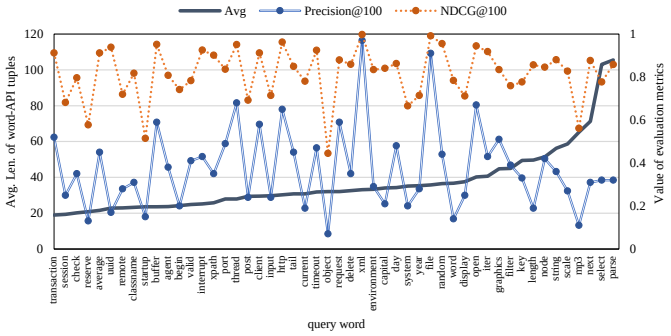


Fig. 4: The average tuple length and the performance.

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.

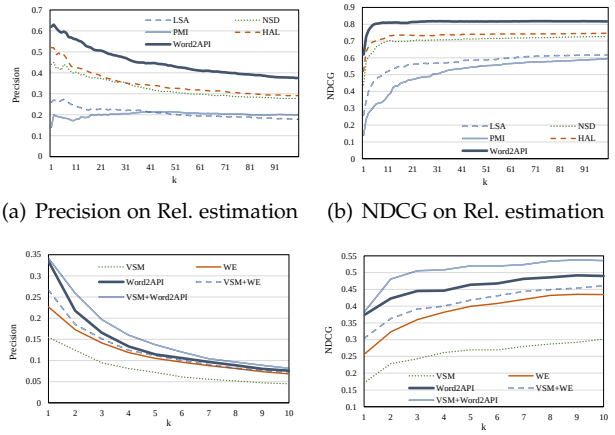
S5 Influence on the Tuple Length

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 Lucene$_{API+Comment}$ proposed in Section 6.4.2. 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$_{API+Comment}$			Word2API$_{Exp}$		
				FR	P@5	P@10	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	*	*	*

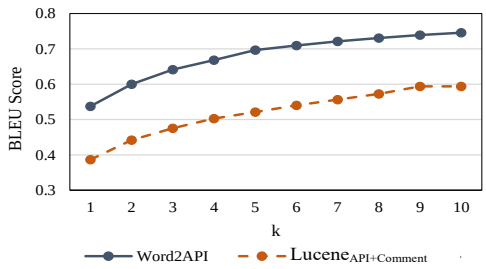


Fig. 6: BLEU score on project-specific search.

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
	$\text{Google}_{Specification}$	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 $\text{Google}_{Specification}$. We find Google provides a strong baseline for information retrieval tasks in software engineering. For API documents linking, the results of $\text{Google}_{Specification}$ 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.