

Complementing Global and Local Contexts in Representing API Descriptions to Improve API Retrieval Tasks

Thanh Nguyen
Iowa State Univ., USA
thanhnng@iastate.edu

Ngoc Tran
Univ. of Texas-Dallas, USA
ngoctran@utdallas.edu

Hung Phan
Iowa State Univ., USA
hungphd@iastate.edu

Trong Nguyen
Iowa State Univ., USA
trongn@iastate.edu

Linh Truong
Univ. of Texas-Dallas, USA
linh.h.truong@utdallas.edu

Anh Tuan Nguyen
Axon Corp, USA
ntanhbk44@gmail.com

Hoan Anh Nguyen
Iowa State Univ., USA
hoan@iastate.edu

Tien N. Nguyen
Univ. of Texas-Dallas, USA
tnn160630@utdallas.edu

ABSTRACT

When being trained on API documentation and tutorials, Word2vec produces vector representations to estimate the relevance between texts and API elements. However, existing Word2vec-based approaches to measure document similarities aggregate Word2vec vectors of individual words or APIs to build the representation of a document as if the words are independent. Thus, the semantics of API descriptions or code fragments are not well represented.

In this work, we introduce D2VEC, a new model that fits with API documentation better than Word2vec. D2VEC is a neural network model that considers two complementary contexts to better capture the semantics of API documentation. We first connect the global context of the current API topic under description to all text phrases within the description of that API. Second, the local orders of words and APIs in the text phrases are maintained in computing the vector representations for the APIs. We conducted an experiment to verify two intrinsic properties of D2VEC's vectors: 1) similar words and relevant API elements are projected into nearby locations; and 2) some vector operations carry semantics. We demonstrate the usefulness and good performance of D2VEC in three applications: API code search (text-to-code retrieval), API tutorial fragment search (code-to-text retrieval), and mining API mappings between software libraries (code-to-code retrieval). Finally, we provide actionable insights and implications for researchers in using our model in other applications with other types of documents.

CCS CONCEPTS

• **Software and its engineering** → **Software maintenance tools**;

KEYWORDS

Word2vec; Big Code; API documents; Code Search; API Mappings

ACM Reference Format:

Thanh Nguyen, Ngoc Tran, Hung Phan, Trong Nguyen, Linh Truong, Anh Tuan Nguyen, Hoan Anh Nguyen, and Tien N. Nguyen. 2018. Complementing Global and Local Contexts in Representing API Descriptions to Improve

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.

ESEC/FSE '18, November 4–9, 2018, Lake Buena Vista, FL, USA

© 2018 Association for Computing Machinery.

ACM ISBN 978-1-4503-5573-5/18/11...\$15.00

<https://doi.org/10.1145/3236024.3236036>

API Retrieval Tasks. In *Proceedings of the 26th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE '18)*, November 4–9, 2018, Lake Buena Vista, FL, USA. ACM, New York, NY, USA, 12 pages. <https://doi.org/10.1145/3236024.3236036>

1 INTRODUCTION

An important class of software engineering (SE) tasks is centered around *the relevance between text descriptions and source code*. An SE application is *text-to-code retrieval*, e.g., helping developers find *code examples* from a given textual query [8, 11, 29, 52, 55]. Another SE application in the other direction with *code-to-text retrieval* is to automatically search relevant textual fragments in an API tutorial for a given API query [22, 23, 41]. In general, a wide range of other SE applications also involves text and code relevance, including bug localization [53, 57], software traceability [12], code summarization [17, 48], bug triaging [5], documentation processing [46, 47], and naming convention [1, 2].

The advances in natural language processing (NLP) have been applied to SE documents and have achieved much success [19]. One of the advanced techniques that have gained much popularity is Word2vec [33]. After being trained on a large corpus of texts, it is capable of projecting terms into a continuous vector space in which the semantically related terms are mapped to nearby locations, thus, it can capture the relevance between texts and APIs [54]. Specifically, Word2vec is run on API documentation and tutorials of software libraries to build the vector representations (called *embeddings*) for the English terms and API elements. Since *documentation and tutorials contain both texts and API elements*, Word2vec is able to *project terms and API elements in a shared vector space*. For example, the API String.substring is mapped to a nearby location with the terms used to describe that API. Then, semantic similarities between two documents are estimated via a weighted sum of the similarities between the words in those documents. Such similarity is used to improve accuracy in code retrieval from a text [50] and in bug localization between bug reports and files [54].

Despite the successes in those SE applications, those retrieval approaches [50, 54] with *Word2vec-based similarity feature* face a key issue in *using the Word2vec vectors of individual words in the documents to estimate document similarities*. The vectors of the words are aggregated as if the words in the same document are treated as independent. In source code, the program dependencies among API elements are important as different dependencies might result in different semantics of a code fragment. In textual documents, treating their words as independent without considering

word orders will not capture well the semantics of the documents, leading to imprecise estimation of document similarities. In those approaches [50, 54], different sentences can have exactly the same representation if the same words are used. Moreover, with the Continuous Bags of Words (CBOW) architecture of Word2vec [33] on API documentation, an API is represented by a vector that is encoded with *the bags of surrounding words* being used in the description of the API. However, in many cases, *e.g.*, for the closely semantically related APIs in the same class, we need a representation that can precisely capture *the sentences in the API's description* so that the functionality of each API can be better represented.

In this paper, we conjecture that we need a new model that best fits with API documentation and SE documents. We present D2VEC, a neural network model that considers two complementary contexts to better capture the semantics of API descriptions. First of all, instead of using bags of words, we realize the local context in which our unsupervised algorithm learns the vector for an API by training on the description of the API to *predict the next word given its previous words and the API topic under description*. That treatment allows the model to consider the local orders of words and related APIs in the description. Second, at the same time, the topic of the API under description plays the role of the global context that *repeatedly connects the API topic to each of the text phrases in the description in the training windows* and the topic serves as a memory of the current topic for all the phrases. In comparison, in the existing approaches [50, 54], the Word2vec vectors are computed in a separate step, and then aggregated to compute document similarity in which no word order is considered. Moreover, while existing models represent an API by the vector corresponding to the API within the text descriptions, in our model, the vector for the API is the vector for the API topic, which is aimed to capture the entire description of the API. Our model is trained to maximize the likelihood of seeing all text phrases in the description of that API as the API topic is attached to all those phrases during training.

We conducted an experiment to verify the intrinsic properties of the vectors produced by D2VEC. Similar to Word2vec, D2VEC's vectors have the following properties: 1) similar words and relevant API elements (*e.g.*, `WordUtils.capitalize` and `StringUtils.toUpperCase`) are projected into nearby locations in the vector space; 2) some vector operations carry meaning. For instance, the vector $V(\text{"append"}) - V(\text{StringWriter.append}) + V(\text{String.concat})$ is closest to the vector of the word *"concatenate"* where $V(\cdot)$ denotes a vector, and the $-$ and $+$ signs denote the vector subtraction and addition, respectively. The interpretation of this operation is that *"append"* explains the functionality of the API `StringWriter.append`, thus, *"concatenate"* can be used to describe the functionality of the API `String.concat`.

We also showed the performance of D2VEC on three SE applications. The first one is API example code search (*text-to-code retrieval*) in which our goal is to retrieve the API code examples given an English query. The retrieval accuracy on a dataset of code examples from KodeJava [25], a tutorial site, is 19.9-41.2% with one to five suggestions. Compared to Word2vec, D2VEC achieves 8.0% and 11.7% higher for top-1 and top-5 accuracies, respectively. Another SE application is *code-to-text retrieval* in which a fragment of an API tutorial is retrieved for a given API as a query. Our experiments on five publicly open tutorials show that with D2VEC, the retrieval is improved over Word2vec by up to 8% in F-score.

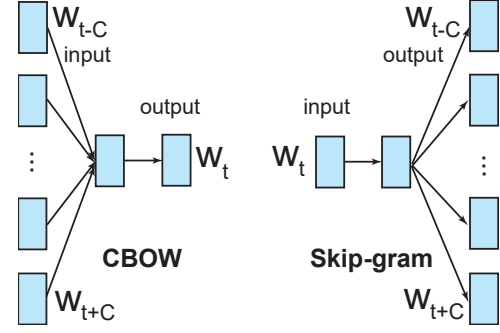


Figure 1: Word2vec with two different architectures

We also evaluate D2VEC's in the application of mining *the mappings between APIs of two libraries*, JDK [21] and Apache [6] (*code-to-code retrieval*). For this, we rely on the principle that two corresponding APIs with the same/similar functionalities in two libraries are explained in similar descriptions. Thus, they are mapped to nearby locations in the vector space. That is, D2VEC's vectors can be used to relate two APIs in two libraries since they capture the respective descriptions. In deriving the mappings for 100 most common JDK APIs, the accuracy with a single suggestion is 36%.

Finally, we provide actionable insights on using D2VEC in the context of other documents, *e.g.*, source code, in other applications.

In this paper, we make the following key contributions:

- 1) D2VEC, that is adapted to fit with API documentation to produce better representations for APIs than existing approaches;
- 2) The idea of using a topic/keyword imposing on all phrases that are semantically relevant to that topic to represent better the content of those phrases, and its applications in other SE tasks;
- 3) Three applications of D2VEC in text-to-code retrieval (API example code search), code-to-text retrieval (API tutorial search), and code-to-code retrieval for the API mappings in two libraries.

2 WORD2VEC FOR API DOCUMENTATION

2.1 Background on Word2vec

Among the models proposed to estimate the vector representations of the words in natural language texts [7, 31–33], Word2vec [33] has recently attracted much attention. Mikolov *et al.* [31, 33] introduce two models, named Continuous Bag-of-Words (CBOW) and Skip-gram (Figure 1), to learn the vector representations for the words from text corpus. Generally, CBOW predicts a center word given other words in a context via a neural network that takes the context words as input and produces prediction probabilities for every target word in the vocabulary V . To do that, CBOW represents each input word as a N -dimensional vector via a matrix $W_{V \times N}^{(1)}$ such that the k -th row of $W^{(1)}$ corresponds to the vector representation of k -th word in V . From the input vectors, the hidden layer output h can be computed by summing these vectors, and another matrix $W_{N \times V}^{(2)}$ in the second layer is used to compute the ultimate probabilities in the output. For Skip-gram, its input is the center word of a window and the output is the surrounding ones. The task can be considered as predicting the context given a word. In addition, more distant words are given less weight by randomly sampling.

Word2vec model is trained with stochastic gradient descent over a large collection of texts. The learned vector representations have two key properties. First, words that are semantically similar have vectors closer to each other in the embedding space. For example, “powerful” and “strong” are closer than each to the vector of “Paris” [27, 33]. Additionally, the difference between word vectors also contains the information on semantic relations. For example, $V(\text{“king”}) - V(\text{“man”}) \approx V(\text{“queen”}) - V(\text{“woman”})$, where V is Word2vec and the minus sign denotes vector subtraction operation.

2.2 Word2vec for Embedded API Elements

Ye *et al.* [54] apply Word2vec with Skip-gram architecture on API documentation and tutorials to learn the vector representations of English terms and API elements including API classes and methods. They treat the API elements embedded within the texts of documentation as special words. The vectors computed for these special words after training are used as the vectors for API elements. Furthermore, for API documentation, they add the fully qualified name of each API to the beginning of the description for that API, and then run Word2vec on all the descriptions. The relevance between API elements and regular words can be considered as parts of the semantic relations among two words, which is successfully captured by Word2vec. Relevant API elements and words often occur with similar words and related API elements (called *surrounding contexts*) in the texts of the API documentation. Those words and code elements are expected to have nearby representations.

Let us take as an example of the API `StringBuilder.substring(int, int)` in Figure 2. The documentation concisely describes the functionality of `StringBuilder.substring`. The sentences and the phrases “subsequence of characters”, “substring”, “contained in this sequence”, etc. in the description explain the functionality of `StringBuilder.substring` as well as the relation to the API `String.String`. A typical characteristic of API documentation that makes the Word2vec model appealing to our problem is the mixture of API elements and English words in the API descriptions. The fully qualified name for the API `java.lang.StringBuilder.substring` is placed at the start of its description and Word2vec model is trained on all the API descriptions to build the embeddings/vectors for the words and the APIs (embedded within the texts). The semantic similarities between two documents are then estimated via a weighted sum of similarities between their words in which no word order is considered [54].

3 D2VEC MODEL FOR API DOCUMENTATION

3.1 The D2Vec Model

In D2Vec, for each API in an API documentation, we aim to build a vector for the textual description of that API. The vector for its description is used to represent the API itself. We aim to capture the semantics of the description of the API as well as keep the word orders in that resulting vector to reflect the description better than the bags of surrounding words. At the same time, we want to associate the API under description with the semantics of its actual description. That is, we aim to have the current API under description as a global topic for all the phrases within that description.

To achieve that, we use two complementary types of context in our model. First, we use the API topic as the **global context** for all the phrases within the description of the API. Specifically, we add

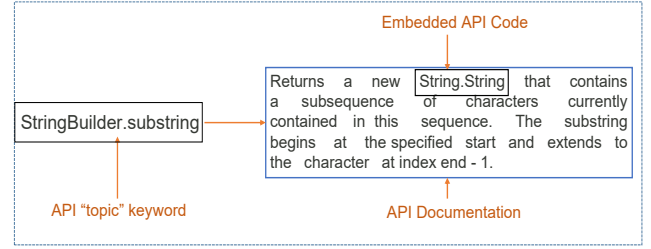


Figure 2: Documentation of JDK API `StringBuilder.substring`

an ID for an API to each of the windows of texts in the description of that API (see Figure 3). The IDs are uniquely assigned for all the APIs of interest in the API documentation in the corpus. The ID token is a special word that acts as memory remembering what is missing from the current context window of words, i.e., the **topic** of the description [26]. The API ID vector is shared across all context windows generated from the description for the same API but not across the descriptions for other APIs. After training, the vector corresponding to an API ID is used as the vector/embedding for the API itself. In other words, D2Vec learns vector representations from contextual information in sentences, as well as incorporating the *topic* of the description, which is *the current API element under description*. The topic, which is the API under description, is always connected to all the phrases/windows in the description and we use it to enrich the representations. That topic connects those words in the same description with the same semantic, which is the API under description. Thus, *the topic is expected to help the model capture the semantics of the API description*.

Second, we use the previous words in the description as the **local context** to preserve word orders. While the semantics of a word in a description is captured in its vector as in Word2vec, an API is represented by the vector for its description considering the word order in small windows, thus preserving word orders locally [26]. The vectors for the words in the descriptions are computed as regular embeddings in the traditional Word2vec. The embedded APIs are replaced with the corresponding API IDs.

Both the topic as the global context and the previous words as the local context provide the clues for the prediction of the next word in all the windows in the description of the current API. That is, the API ID vector and the vectors of the words/APIs in the descriptions are combined to predict the next word in a context window. The vector for the API ID (representing its entire description) can be used as features for the API in retrieval algorithms.

3.2 Architecture

Figure 3 displays the architecture. Mathematically, given a description $w_1, w_2, \dots, w_T$ of the API a , we maximize the log probability:

$$L = \frac{1}{T} \sum_{i=C}^{T-C} \log p(w_i | w_{i-C}, w_{i-C+1}, \dots, w_{i-1}, a)$$

The conditional probability $p(w_i | w_{i-C}, \dots, w_{i-1}, a)$ and the prediction task are modeled via a multiclass classifier with the network in Figure 3 and a softmax function. The word prediction task considers the given **local context** and the **global topic** on which this context covers. Each of y_i for each output word i is un-normalized

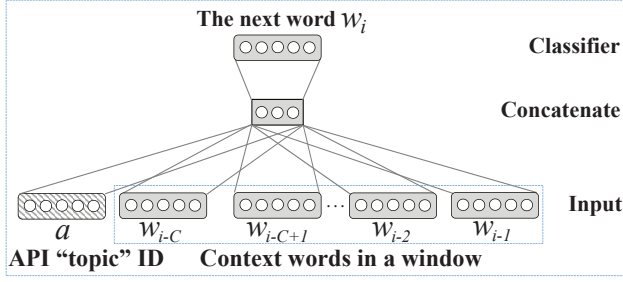


Figure 3: D2VEC model to learn vector representations for words and API elements from API documentation

log-probability and is computed as:

$$y = b + Uh(a : A; w_{i-C}, \dots, w_{i-1} : W)$$

where U and b are the softmax parameters. We use W to denote the word matrix being formed by putting together all word vectors as its columns. A is called the API ID matrix that is formed by putting together all unique vectors for all the API IDs. The h function is constructed by a concatenation of the word vectors extracted from the matrix W and the API ID vectors extracted from the matrix A . Note that the word orders in w_i are considered.

Training for D2VEC is done with stochastic gradient descent similarly to PVDV's and Word2vec's [26, 33]. At each step of stochastic gradient descent, a fixed-length context is randomly sampled from a description. We then compute the error gradient from the network and use the gradient to update the model's parameter. At prediction time, for a new text, the inference step is also done via gradient descent where the rest of the parameters of the model, e.g., all the word vectors and the softmax weights are fixed.

D2VEC learns vector representations with two key advantages: 1) it enriches information of the API elements that do not occur within the documentation descriptions of other API elements; 2) the API elements with similar descriptions are likely to be close to each other in the vector space. Properties from Word2vec such as the vector offsetting still hold for D2VEC (will be explained later).

3.3 Approach Applicability and Implications

In this work, the topic is the API under description and the training data is textual documentation. However, the direction that we put forth here is applicable to other software engineering tasks related to textual documentation and source code. Let us explain how we would extend the idea of topic for other SE tasks.

3.3.1 Representation for a Code Fragment with an Addition of a Topic/Keyword. If we apply our idea to source code and inline comments, we could view the *text comments as the topics for the source code fragments relevant to those comments*. If we extract keywords or topic words from the comments and use D2VEC on source code, we could learn the vector representations that capture the entire code fragments as well as the relations among the words in the comments and code elements. For example, in Figure 4, we could use the inline comment "Converts back the list into array object and prints the new values" as the text relevant to the code below it. The training data is the set of source files that have high-quality comments for source code. We could first extract the keywords in the

comment such as "convert", "list", "array object", "print", "value" and use them as the topics for the code elements in the code fragment below the comment. We train a model to predict the next code token or API in that code fragment. As a consequence, the model will be able to capture the sequence of code or API elements that are relevant to the topics/keywords. In brief, in this representation, the topics are English keywords to describe the semantics for the sequence of code tokens or API elements in a code fragment.

An application of such a model is API code search (Section 5): a user gives a query or a keyword, the model will rank the list of code fragments depending on their relevancy levels to the query.

3.3.2 Applying D2VEC on Bug Localization with LDA. Let us discuss D2VEC in the context of bug localization [54] in which the text in a bug report is given, and the model will rank the source files relevant to the report for the debugging purpose. In bug localization, the topics could be built from the bug reports by extracting keywords or using topic models such as LDA [9]. We could use the bug reports and the corresponding fixed files as training data. Finally, the other similar SE tasks include software traceability from texts to code, and code summarization (summarizing source code with texts).

3.3.3 Database of Vector Representations for APIs as Building Blocks. For software libraries, their APIs could be described in many sources: API documentation, tutorials, online discussions, inline comments, and their source code. For each kind of documents, we could train D2VEC to produce the representations for all the APIs of the libraries of interest. For example, we could produce the vectors for `String.substring` by running D2VEC on JDK documentation for that API, running on CodeJava tutorial and on its source code in JDK.

An interesting research question is to compare and combine such the distributed vector representations of the APIs to achieve a better retrieval mechanism for SE applications. For example, the vectors produced by running on API documentation would have different characteristics with the vectors on source code. However, they both capture the semantics/relevance of the API of interest to the text description and its implementation source code. One could expect to exploit those two types of vector spaces to support the SE tasks related to linking API documents and source code.

4 RESEARCH QUESTIONS AND DATASET

4.1 Research Questions

We conducted several experiments to evaluate D2VEC in SE applications that involve text-to-code retrieval, code-to-text retrieval, and code-to-code mappings. We focus on the following questions:

RQ1. How does D2VEC compare or help improve a baseline retrieval model in the task of API code search (text-to-code retrieval)?

RQ2. How well does D2VEC perform in the task of searching tutorial fragments relevant to a given API (code-to-text retrieval)?

RQ3. How well does D2VEC perform in an application of finding the mappings for the APIs in two libraries (code-to-code retrieval)?

4.2 Training Corpus

We collected a dataset of the API documentation of Java JDK Core and Apache Commons libraries (we call them JDK and Apache for short) [6, 21] to train the models to learn the vector representations of words and API elements. Table 1 shows the dataset's statistics.

Table 1: JDK and Apache Training Dataset

Number of distinct API elements	36,923
Number of distinct English words	10,258
Occurrence frequency of APIs	3.2
Occurrence frequency of English words	25.6
Vocabulary size	47,381
Number of APIs embedded within a documentation	2.7

In addition to getting textual descriptions, a non-trivial task of building training data is to identify and annotate API code embedded within the descriptions. Fortunately, JDK and Apache open-source files are available and Javadoc style comments can be parsed using the Java development toolkit (JDT) to accomplish the task.

The procedure for data collection was as follows. We first collected Java source files of both JDK and Apache libraries. Then, we parsed each source file with JDT to obtain the declarations (*i.e.*, API fields/methods/classes) and Javadoc-style comments. These documentation comments are stored in AST nodes that were accordingly traversed using some specific tags such as “@code”, “@param”, “@return” to identify API elements. After that, we annotated the identified API code elements with their packages and class names (*e.g.*, `java.lang.StringBuilder.append`) if fully qualified names are available from traversing the AST nodes; otherwise, we utilized a dictionary of JDK and Apache APIs. We considered all annotated APIs as special API IDs in the documentation and added them to a vocabulary for training the models. For D2VEC, we also labeled the descriptions with their own APIs (*i.e.*, the “topic” words). We removed stopwords and stemmed the remaining words. Note that, for the traditional Word2vec, the fully qualified name of an API is added to the beginning of its text description and the model was run on all the descriptions. For D2VEC, an API ID is added to all the context windows for all phrases of the description of the API.

Finally, we have the API documentation corpus with 36,923 distinct APIs and 10,258 distinct English words (Table 1). On average, each API occurs 3.2 times in the entire corpus and each documentation contains 2.7 API elements. The occurrence frequency for English word is 25.6. Compared with the English dictionary, the number of words in this dataset is much smaller. Thus, this relatively high regularity suggests that the dataset would be helpful in learning vector representations via neural network-based models.

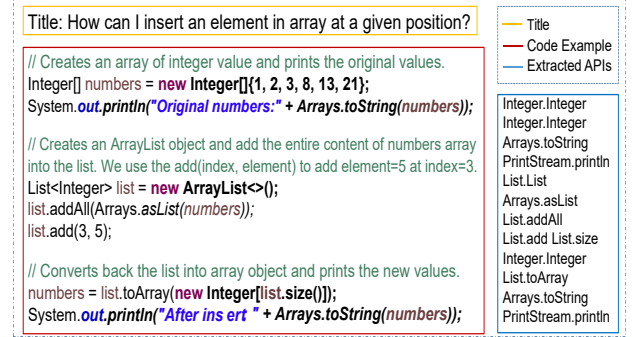
5 RQ1. API EXAMPLE CODE SEARCH

This section presents an evaluation of our model on an important text-to-code retrieval application, API example code search [11]: given an English query, a tool returns the code examples in a code-base that are semantically relevant the most to the query. We used the dataset in Table 1 to train D2VEC to obtain the vectors for the APIs. For an example, we extracted the sequence of API elements and used the trained model to build the vector for that sequence to represent the code example. For the query, we used our trained model to run on the words of the query to produce the vector representation for it. Then, we used cosine similarity scores between two documents to rank the API code examples for a given query.

We also compared our model against the Word2vec Skip-gram model used in the work in Ye *et al.* [54] and the revised Vector Space Model (rVSM) [57]. We chose the first one because it is a

Table 2: KodeJava Test Dataset for Retrieval

Number of API code examples	437
Number of unique words	644
Number of unique APIs	986
Word occurrence frequency	6.7
API occurrence frequency	4.3
Average length of queries	9.8
Average length of code examples	11.9

**Figure 4: An API code example with query from KodeJava**

state-of-the-art neural network-based model, while the second one is an advanced vector representation model that has been shown to outperform other traditional information retrieval models [57]. For Word2vec, the vectors for a query or a code example are computed by aggregating the vectors of all the words in the query or all the APIs in the example in the same way as explained in their paper. For rVSM, a query and a code example are represented by the Tf-Idf vectors considering the words and the APIs in the example.

5.1 Data Collection

5.1.1 Training Data for Retrieval. For this experiment, we built training data to include only the documentation of JDK APIs in Table 1. Finally, we have the dataset with 15,712 unique API elements and 5,679 English words. The occurrence frequencies for an API and a word are 3.1 and 22.2 respectively.

5.1.2 Test Data for Retrieval. To evaluate the performance of our model, we collected a parallel dataset of queries and API code examples from a Java tutorial website, named KodeJava [25]. Each KodeJava post contains a title that describes a programming task and the JDK API code fragment(s) to fulfill that task. For code retrieval, we utilized the title and the code fragment with the highest score as a ground truth. An example is shown in Figure 4 in which a developer places a query on how to insert an element in an array given a specified position. For retrieval, we processed code examples to extract API elements using the JDT parser (Figure 4). Finally, our dataset (Table 2) contains 437 pairs of titles and corresponding sequences of API code elements from the corresponding examples.

5.2 Metrics and Setting

For D2VEC and Word2vec, we set the context window size $C = 5$ and the dimensionality $N = 200$ to learn the vectors. For each query, we rank all the code examples. If the correct example occurs in the

Table 3: Accuracy comparison on API code search

	top-1	top-2	top-3	top-4	top-5
Word2Vec	11.7	18.5	22.7	26.3	29.5
D2Vec	19.9	28.8	33.9	38.2	41.2
revised Vector Space Model (rVSM)	19.1	23.0	25.4	27.4	35.2
rVSM+ Word2Vec	28.2	30.4	33.5	34.6	38.5
rVSM+D2Vec	35.5	46.7	52.7	58.1	60.9

Table 4: Example results of API code search and ranks

Q: Question	D2Vec	D2Vec + rVSM
A: Expected APIs		
Q: Breaks a text or sentence into words?		
A: Locale BreakIterator.getWordInst... BreakIterator.setText	1	1
Q: Insert an element in array at a given position?		
A: List.List Arrays.asList List.add List.add List.toArray...	4	1
Q: Get the maximum number of concurrent connections?		
A: Connection.getMetaData Database...getMaxConnections	5	1
Q: Launch user-default web browser?		
A: URL.create Desktop.getDesktop Desktop.browse...	2	1

top- k list of the retrieved examples, we count it as correct; otherwise, it is a miss. Top- k accuracy for API code search is computed as the ratio of the number of hits over the total number of cases.

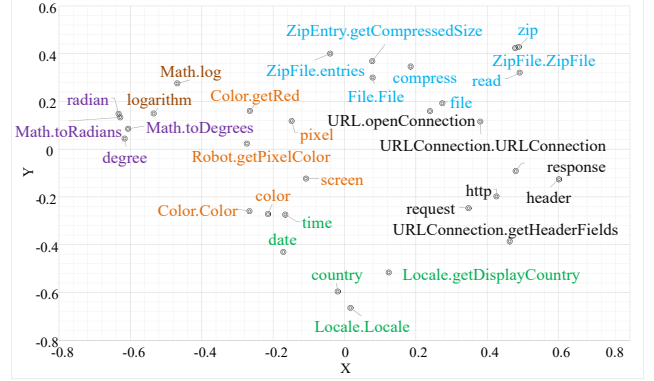
5.3 Empirical Results and Analysis

Table 3 shows the top-ranked accuracies on code retrieval for different approaches. First, the top-1 and top-5 accuracies of Word2vec and D2Vec are 11.7–29.5% and 19.9–41.2%, respectively. As seen, D2Vec improves over Word2vec 8.2% (70% relatively) and 11.7% (39.6% relatively) in top- k accuracies.

To further study D2Vec’s ability to connect queries and code examples that are lexically mismatched, we combine it with an advanced Vector Space Model, rVSM [57]. The relevance score is a linear combination of the two scores from D2Vec and rVSM: $\alpha \times rVSM(Q, C) + (1 - \alpha) \times D2Vec(Q, C)$. We adjust the parameter α based on the Jaccard similarity between the query Q and the code example C . If Jaccard score is greater than the threshold of 0.65, we use $\alpha = 1$ (favoring rVSM). Otherwise, $\alpha = 0$ since we expect D2Vec to complement well for the lexical-mismatch cases. All scores are normalized over all examples. As seen in Table 3, the combined approach **improves 15.6–16.4% top-1 and 19.7–25.7% top-5 accuracies** over the individual approach in rVSM and D2Vec.

In comparison to rVSM, D2Vec performs slightly better because it handles better the cases of lexical mismatches between the queries and source code. Table 4 shows a few examples with lexical mismatches. For instance, for the query “insert an element in array at a given position”, D2Vec and D2Vec+rVSM retrieve the correct code example at high positions. rVSM ranked it at the 22nd place due to the mismatch of “insert” and List.add. Among 87 examples that were ranked first by D2Vec, 35 cases were not retrieved by rVSM in the first 20 results due to low textual similarities between queries and code examples. Thus, D2Vec *complements well to rVSM in the cases with low textual similarities* between a query and source code.

Finally, the combined model rVSM+D2Vec outperforms rVSM + Word2vec. Thus, D2Vec helps improve better the baseline model rVSM than Word2vec-based similarity measure in Ye *et al.* [54].

**Figure 5: 2-D visualization of API and word vectors via PCA****Table 5: Example of relations via vector offsets in JDK**

R1. Get the next element

StringTokenizer.hasMoreTokens StringTokenizer.nextToken
Scanner.hasNext Scanner.next
HashMap.containsKey HashMap.get
Stack.isEmpty Stack.pop

R2. Check the size of a collection before getting an element

java.util.ArrayList.size java.util.ArrayList.get
java.util.Map.size java.util.Map.get

R3. Insert an element

java.util.Vector java.util.Vector.add
java.util.TreeMap java.util.TreeMap.put

R4. Check before adding an element

LinkedList.contains LinkedList.add
Hashtable.containsKey Hashtable.put
Set.contains Set.add

R5. Write to and flush a stream

FileOutputStream.write FileOutputStream.flush
PrintWriter.write PrintWriter.flush

5.4 Properties of Vector Representations

5.4.1 Nearby Vectors of Words and Related API. We conducted another experiment to examine the arrangements of the vectors for English words and related API elements. We selected English words and API elements from six retrieval ground truth pairs of queries and code examples from KodeJava dataset (Table 2). Then, we projected their vectors onto the 2-dimensional space for visualization with Principal Component Analysis (PCA) [24] (Figure 5).

As seen, English words and API elements that are grouped into nearby locations in the vector space belong to the same ground truth. For example, the words “zip”, “compress” and “file” are clustered more closely to File.File, ZipFile.entries, ZipEntry.getCompressedSize than to the other elements. Similarly, “radian” and “degree” are relatively close to Math.toRadians and Math.toDegrees. In brief, D2Vec projects the words and relevant API elements into nearby locations in the vector space. Thus, it allows our tool to highly rank the code examples with the APIs relevant to the query’s terms.

5.4.2 Nearby Vectors of Related APIs. We observed that the APIs with related functionalities are also projected to nearby locations. Those APIs are often described/surrounded by similar words, thus,

they have nearby vectors. For example, `OutputStreamWriter::write`, `PrintStream::printf`, `BufferedWriter::flush` have nearby vectors. As an implication, we could use the vector space to find relevant APIs.

5.4.3 Vector Offsets and Relations Between Words and API Elements. Vector offset operations hold true to the D2Vec vector representation. For instance, $V(\text{"read"}) - V(\text{BufferedReader.read}) + V(\text{BufferedWriter.write})$ becomes a vector closest to the vector representation of the word "write". The interpretation of this operation is that the word "read" explains the functionality of `BufferedReader.read`, thus, "write" could be used to describe `BufferedWriter.write`. Another example is $V(\text{"append"}) - V(\text{StringBuilder.append}) \approx V(\text{"concatenate"}) - V(\text{String.concat})$. That is, *this operation captures a word being used to describe the behavior of an API element.*

5.4.4 Vector Offsets and Relations Between APIs. We observed that vector offsets could explain the relations between APIs. For example, $V(\text{Scanner.hasNext}) - V(\text{Scanner.Scanner}) + V(\text{StringTokenizer.StringTokenizer})$ is close to the vector of `StringTokenizer.nextToken`. That is, using `hasNext` on a `Scanner` has the same meaning as using `nextToken` on a `StringTokenizer`. Using our knowledge on JDK APIs and our PCA visualization (not shown), we have observed several relations via the vector offset operation (Table 5). For example, some pairs have different names for the same function (R4 in Table 5).

6 RQ2. SEARCHING RELEVANT FRAGMENTS IN API TUTORIALS

In this experiment, we evaluate D2Vec on a code-to-text retrieval task: retrieving a text fragment in a tutorial of a library that is relevant to a given API. Learning to properly use API is important. A practical way is to learn from a tutorial. However, such a tutorial is usually lengthy and mixed with irrelevant information. It is tedious for developers to peruse a full tutorial for an unfamiliar API [23].

A challenging question is to determine which fragments are relevant, *i.e.*, containing explanatory information to developers. A fragment of text in a tutorial might contain the names of the other "related" APIs, but not explanatory information on API usages [23]. Non-explanatory sentences contain the APIs for an overview of an entire API class or an enumeration of related APIs.

Specifically, this problem is stated as follows. A user provides the name of an API in which (s)he is interested. The task is to rank the fragments in a tutorial with respect to containing explanatory information relevant to a given API as a query. A tutorial contains several fragments/paragraphs of texts (with mixture of APIs). Figure 6 shows an example of a fragment in Android Graphics Tutorial. There are four APIs that appear in the fragment: `Canvas`, `Bitmap`, `SurfaceHolder`, and `SurfaceView`. According to the manual annotation [23], `Canvas` and `Bitmap` are relevant to this fragment, whereas `SurfaceHolder` and `SurfaceView` are not since the fragment is not about those 2 types of objects. Several approaches have aimed to address this problem using information retrieval [22, 23, 41].

6.1 Data Collection

For a comparative study, we chose to use the state-of-the-art tool FRAFT [23] with a dataset and the ground truth used in its experiments. We used this dataset of five libraries as test data (Table 6). To train the models to build vectors, we downloaded the source

(1)When you're writing an application in which you would like to perform specialized drawing and/or control the animation of graphics, you should do so by drawing through a **Canvas**. (2) A **Canvas** works for you as a pretense, or interface, to the actual surface upon which your graphics will be drawn. (3)It holds all of your "draw" calls. (4)Via the **Canvas**, your drawing is actually performed upon an underlying **Bitmap**, which is placed into the window. (5)In the event that you're drawing within the `onDraw()` callback method, the **Canvas** is provided for you... (6)You can also acquire a **Canvas** from `SurfaceHolder.lockCanvas()`, when dealing with a **SurfaceView** object. (7)However, if you need to create a new **Canvas**, then you must define the **Bitmap** upon which drawing will actually be performed. (8)The **Bitmap** is always required for a **Canvas**. (9)You can set up a new **Canvas**...

Figure 6: A fragment in Android Graphics Tutorial

Table 6: Tutorial Corpus

Tutorial	#APIs	Explanatory Fragment	Non-Expla. Fragment	Frag with APIs	Frag w/o APIs
Joda Time	36	19	10	21	8
Math Lib	73	31	10	16	25
Col. Official	59	31	26	17	40
Col. Jenkov	28	34	35	42	27
Smack	40	42	5	31	16

code including the Javadoc of those libraries. We parsed the code and Javadoc to produce the documentation for the APIs in those libraries, which was used to train the models to build the vectors.

6.2 Approach using D2Vec

Let us explain how we use D2Vec for this problem. The intuition of using D2Vec is that the fragment on a specific API needs to be semantically relevant to a high degree to the description of the API in API documentation. They might not be exactly matched but must semantically relevant. Specifically, we used D2Vec to compute the relevance score $\text{sim}(A, F)$ between an API A and a fragment F . To compute the score, we trained D2Vec with the Javadoc API documentation of the 5 libraries included in their corresponding source code as explained earlier. After this step, we have the vectors V_{APIs} for all the APIs in those five libraries. For example, in Figure 2, the vector for the paragraph on the right side is the vector for the API `StringBuilder.substring`. The vector V_F for each fragment F in a given API tutorial is computed during that process. The relevance score $\text{sim}(A, F)$ is computed as the cosine similarity between the vector V_A of the API A after training and the vector V_F for the fragment F .

6.3 Metrics and Setting

In this experiment, we compared D2Vec against Word2vec and FRAFT [23], which is a state-of-the-art information retrieval approach (FRAFT contains two parts: fragmenting the tutorials and searching relevant tutorial fragments to which we compared). Because we focus only on comparing the searching part, we used the fragments provided as part of the ground truth from FRAFT's authors [23]. We used each of the three techniques, Word2vec, FRAFT, and D2Vec to rank the fragments and measured the accuracy.

We used the same setting, dataset, oracle, parameters, and metrics as in FRAFT [23]. Specifically, precision is the ratio between the number of correctly predicted relevant *fragment-API pairs* over all the retrieved pairs. Recall is the ratio between the number of the correctly predicted relevant *fragment-API pairs* over all the pairs. F-score is the harmonic mean between precision and recall.

Table 7: Accuracy in Ranking Relevant Tutorial Fragments for APIs

	Precision (%)			Recall (%)			F-score (%)		
Tutorial	FRAFT	Word2Vec	D2Vec	FRAFT	Word2Vec	D2Vec	FRAFT	Word2Vec	D2Vec
Joda Time	85.19	44.62	73.17	76.67	100.00	100	80.70	61.70	84.51
Math Lib	84.78	81.25	82.81	73.58	98.11	100.00	78.79	88.89	90.60
Col. Official	62.03	50.00	64.29	87.50	58.93	64.29	72.59	54.10	64.29
Col. Jenkov	61.19	48.57	57.69	97.62	100.00	88.24	75.23	65.38	69.77
Smack	77.94	87.88	87.88	94.64	100.00	100.00	85.48	93.55	93.55
Average	74.23	62.46	73.17	86.00	91.41	90.50	78.56	72.72	80.54

Table 8: Difference between FRAFT's and D2Vec's results

Tutorial	FRAFT _{Correct}	Both _{Correct}	D2Vec _{Correct}
Jodatime	0	20	10
Math Lib	0	39	14
Col. Official	16	33	3
Col. Jenkov	2	15	0
Smack	0	27	2
Total (# correct cases)	18	134	29
As Percentage (%)	13.43	100	21.6

6.4 Empirical Results

As seen the results in Table 7, D2Vec achieves precision, recall, and F-score of 73.2%, 90.5%, and 80.5%, respectively. Comparing with Word2vec used in Ye *et al.* [54], D2Vec has **much better precision (73.2% versus 62.5%), while maintaining similar recall**. Thus, that resulted in an improvement of nearly 8% in F-score. Since D2Vec takes into consideration both the orders of words and context of the API's description as a whole, the generated vectors are under more constraints to represent the APIs comparing to the Word2vec vectors. Since D2Vec has both local and global contexts, it can predict better, leading to much increase in precision.

D2Vec also achieves slightly better results than FRAFT. In particular, D2Vec increases recall by 4.5% while losing only 1% precision, leading to an overall increasing F-score of 2%.

We further conducted another study to investigate how many cases that D2Vec correctly predicted while FRAFT did not and vice versa. As seen in Table 8, D2Vec can correctly predict 21.6% of the cases that FRAFT did not. Meanwhile, FRAFT can correctly predict only 13.4% of the cases that D2Vec did not. In brief, there is 35% difference in results between the two tools. Therefore, two approaches complement well to each other, leading to a promising direction of combining them to achieve better accuracy.

7 RQ3. MAPPING BETWEEN API LIBRARIES

Inspiring by the properties of D2Vec's vectors in Sections 5.4.1 and 5.4.2, we present a code-to-code retrieval application of D2Vec in library migration: deriving single mappings for equivalent APIs between the two libraries, JDK and Apache.

In software development, an application could appear in multiple platforms. Each platform might require the use of different software libraries. While companies like to develop their software in multiple platforms with different libraries at the same time, they often develop software in one environment and then migrate it to

another using a different library. This process is called *library migration*. For migration, an important task is to find the respective APIs with the same/similar functionality in two libraries. For example, one uses the JDK API `String.toUpperCase` to capitalize the characters of a string. (S)he could also use `StringUtils.capitalize` in Apache. The process of manually defining API mappings not only requires much effort, but also is error-prone and incomplete [56].

To reduce manual effort, several approaches to automatically map API elements have been proposed [35, 37, 40, 56]. Traditional approaches use *heuristics* to map the API classes and methods in two libraries based on the similarities of APIs' names, parameters, or calling structures [30, 51, 56]. However, APIs with same functionality in two libraries generally could have different names, different parameter types, and even different syntactic types (e.g., a method call is mapped to a field access or array access). Instead of heuristics, Gokhale *et al.* [15] use the execution traces of two similar GUI applications to derive the mappings. It is limited to GUI libraries. IBM Model [10] with statistical learning [35] was proposed to learn the mappings from the corresponding sequences of APIs in the client code of two libraries. However, collecting a parallel corpus of client code is not trivial. Nguyen *et al.* [38] propose an approach to mapping APIs using Word2vec. Nonetheless, it requires a training set of already-mapped pairs between two libraries.

7.1 API-to-API Mapping with D2Vec

Our intuition for API mapping problem is that *if in two libraries, two APIs have the equivalent functionality (i.e., they are the mapped API of each other), they are likely to be described with similar words in their respective descriptions in API documentation*. Section 5.4 shows that D2Vec can capture the relevance among words, APIs, and related ones via the distances and the offsets of their representation vectors in the shared vector space. Thus, it motivates us to evaluate how well D2Vec can represent the mapping between two API elements in two different libraries via their respective vectors when they are equivalent in two libraries. That is, given the API mapping pair of A and A' in two libraries L_S and L_T , the vector of A is closer to the vector of A' than others. Thus, D2Vec could help derive API mapping pairs based on the relative vector distances.

7.2 Evaluation on API Mappings

7.2.1 Data Collection. To train D2Vec to produce the vectors, we used the same training data with both API documentations of Java JDK Core and Apache Commons libraries listed in Table 1. As for testing, we used the oracle API mappings provided in [38].

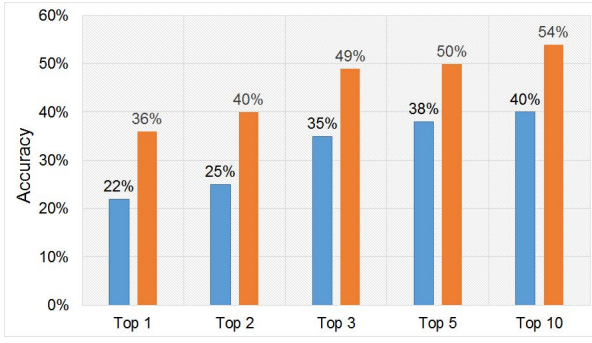


Figure 7: Accuracy comparison in deriving API mappings

Table 9: API Mappings between JDK and Apache by D2Vec

JDK API	Apache API	Rank
String.toUpperCase	WordUtils.capitalize	1
	WordUtils.capitalizeFully	2
	StringUtils.toUpperCase	5
String.toLowerCase	StringUtils.toLowerCase	1
	StringUtils.toLowerCase	5
	WordUtils.toLowerCase	8
Integer.parseInt	NumberUtils.createInteger	10
Math.max	FastMath.max	1
Enumeration.nextElement	IteratorEnumeration.hasMoreElements	2
InputStream.read	AutoCloseInputStream.read	1
	IOUtils.read	2
	TeaInputStream.read	5
Pattern.matcher	StrMatcher.charSetMatcher	1
	StrMatcher.trimMatcher	2
	Matcher.Matcher	4
BufferedReader.readLine	CRLFLineReader.readLine	1
Map.containsKey	AbstractHashMap.containsKey	1
Set.contains	TreeBidiMap.containsKey	1
	SingletonMap.isEqualKey	10
	StringUtils.replacePattern	1
String.replaceAll	StrBuilder.replaceAll	2
	StrBuilder.replaceFirst	3
	StringUtils.replaceEachRepeatedly	9
	FileDeleteStrategy.deleteQuietly	1
File.delete	FileDeleteStrategy.doDelete	2

7.2.2 Metrics and Setting. We used the trained model to derive API mappings as follows. Given an JDK API, we computed its vector via D2Vec. Then, we computed the list of Apache APIs that have the closest vectors to the vector of the JDK API. The Apache APIs are ranked based on their cosine distances to the vector of the JDK API. Note that, we derived only the mappings between API methods in JDK and Apache since the functionality of their methods are well-defined and more verifiable as equivalent than classes (the classes in those two libraries often contain extra functionality). Between Java JDK Core and Apache Commons libraries, an API in one library can be mapped to one or multiple alternatives in the other library. For example, JDK API method `java.io.File.delete` is used to delete the file or folder. This task can also be achieved by using either Apache API `deleteQuietly` or `doDelete` of the class `org.apache.commons.io.FileDeleteStrategy`. Thus, if one of the alternatives is found, we still consider it as a correct suggestion.

We measured top- k accuracy in deriving API mappings. If the candidate list of Apache APIs with k elements contains at least one Apache API that can be used to replace the given JDK API according to the usage documentation in two libraries, we consider the case

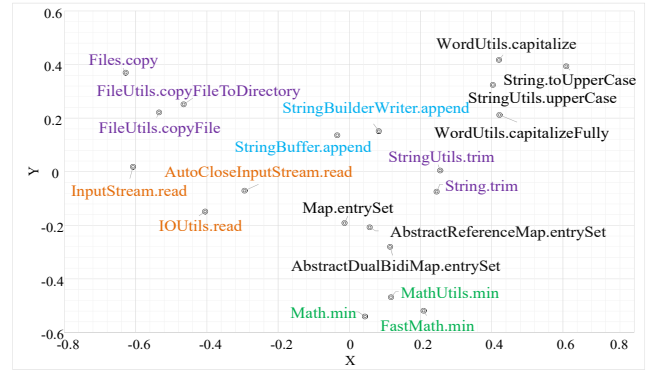


Figure 8: 2-D visualization of JDK and Apache API vectors

as a hit. Otherwise, it is a miss. Top- k accuracy is computed as the ratio between the number of hits over the total number of cases. Accuracy shows how likely one would find the correct mapped API in the resulting list of k Apache API results.

7.2.3 Empirical Results. As seen in Figure 7, D2Vec can derive correct answers for 36 out of 100 cases with only one suggestion. In half of the cases, the correct mappings are in the list of 3–5 suggestions. Note that in many cases, our approach can map more than one correct answers, so the total number of derived mappings for 100 JDK APIs is 98. However, to calculate the top- k accuracy, we only use the best suggestions. As compared to Word2vec, D2Vec outperforms with 14% at top-1 and top-10 accuracies.

Table 9 shows some examples. Some mappings have different names, e.g., `String.toLowerCase` and `StringUtils.toLowerCase`; and `Integer.parseInt` and `NumberUtils.createInteger`. We also reported the rank of the correct API(s) in the ranked list. As seen, D2Vec is able to rank very high the correct Apache APIs for the given JDK APIs.

7.2.4 A Visualization Study. To further study D2Vec’s vectors for JDK’s and Apache’s APIs, we first selected 7 most commonly used JDK APIs and its well-known respective Apache APIs. Then, we projected their vectors into a 2-dimensional space for visualization. Figure 8 shows the geometric arrangements of their D2Vec’s vectors in the 2-D space. The visualization indicates that JDK API and Apache alternatives are mapped to closer locations than other elements. For example, `Math.min` in JDK is close to `MathUtils.min` and `FastMath.min` in Apache; `Files.copy` and `FileUtils.copyFile`, `FileUtils.copyFileToDirectory` are close to each other. Interestingly, the pairs with different names are also captured such as `String.toUpperCase` in JDK and `WordUtils.capitalize` and `WordUtils.capitalizeFully` in Apache. This result shows the reason that D2Vec performs well in deriving API mappings between JDK and Apache as it overcomes the textual mismatch issue between the names of the APIs in two libraries.

8 THREATS TO VALIDITY

For the study on the properties of the vector representations, we do not have a benchmark to verify against. We observed the nearby and vector offset phenomena through vector operations and 2-D projection with PCA. In the experiments for code search application, our KodeJava dataset might not be representative. However, they

were used in a prior code search research [11]. The combination between two approaches is a simple linear combination. However, our goal is to compare with the traditional Word2vec to see how Word2vec and D2Vec improve an baseline model in rVSM.

For API mapping application, our dataset on API mapping pairs might not be representative. However, we picked only the most commonly used APIs among JDK and Apache. In the tutorial search application, the dataset of five libraries might not be representative. However, we used the same dataset as in the FRAFT paper [23].

9 RELATED WORK

D2Vec is related to PVDm [26], which aims to represent phrases, sentences, paragraphs, or documents with vectors via associating IDs to paragraphs. First, D2Vec aims to combine global context (API topic) and local context (n -gram) to represent API elements and the words in API descriptions, which was inspired by the n -gram topic model [39]. In n -gram topic model, the probability that a token c appears at a position is estimated simultaneously based on the global topic k and the local sequence of $n - 1$ previous tokens. While it relies on Bayesian network, D2Vec uses neural network.

Second, D2Vec's architecture and computation are different. In D2Vec, an API topic word is associated with each context window in API documentation and that API word could also occur in the same or different window. That is, the surrounding words of the APIs in API documentation are also considered. Thus, both global context (topic) and local context (n -gram) are combined. In PVDm, a paragraph ID is an ID for a phrase/sentence/paragraph. It is uniquely assigned to identify paragraphs, and does not appear in texts of any paragraph. If paragraph IDs are used to represent APIs, PVDm will not have the local contexts for those APIs. Third, to assign paragraph IDs, PVDm faces the problem of splitting up phrases/sentences/paragraphs, while D2Vec does not need such notions. To empirically compare with PVDm, we need to solve that non-trivial splitting task due to the embedded APIs. Fourth, D2Vec focuses on combining global and local contexts, instead of focusing on only word ordering, which is a derivation of local context via n -grams. Finally, the idea of combining topic modeling and local contexts set forth by D2Vec is applicable to other SE tasks.

Ye *et al.* [54] use Word2vec to project words and code elements into the same space to improve *retrieval accuracy* in bug localization problem. Nguyen *et al.* [50] use Word2vec on API documentation for code retrieval in the same manner. Allamanis *et al.* [2] propose a neural probabilistic language model to suggest method/class names. Code tokens with statistical co-occurrences are projected into a continuous space together with the text tokens from the names. They also introduce a jointly probabilistic model short utterances and code snippets [4]. The bimodal modeling treats texts and code in two distinct spaces. We use one single space.

Guerrouj *et al.* [17] use the context of the words that surrounds a code element in StackOverflow posts to summarize its use and purpose. They follow the spirit of n -gram language model by calculating the frequencies of all the phrases from 1-grams, 2-grams, up to n -grams with certain n value. Then, they use the K -means clustering to select words to include in the summary of an identifier. In comparison, they follow a counting-based approach, while in D2Vec, we use a neural network model.

Information retrieval (IR) is a very common approach in linking code and texts, and searching API usage examples from texts. Some IR-based approaches match queries against the names of components [20], program elements (Sourcerer [8], Griddle [43]), graphs of API elements [11], and relations among API elements [29, 55].

RecoDoc [13] links code-like terms in documentation to their source code elements by using regular expressions to extract them from free-form text and uses partial program analysis for extracting from code. Baker [49] is a tool that incorporates links to the API documentation into the code snippets that use the API. Yang and Tan [52] propose SWordNet to automatically infer semantically related words in software by leveraging the context of words in comments and code. SWordNet does not aim to handle API or program elements. It focuses on the relations between words in order to expand queries with semantically related words.

Statistical NLP approaches have been used in SE. Maddison and Tarlow [28] present a generative model for source code that is based on AST structures. TBCNN [34] also uses trees for suggest next code tokens. Other SE applications of statistical NLP include code suggestion [19], code convention [1], name suggestion [2], API suggestions [45], code mining [3], type resolution [42], etc.

Statistical NLP approaches were used to generate code from text. SWIM [44] first uses IBM Model with word translation to produce code elements. It then uses syntactic rules on those elements to build code sequences close to the query. DeepAPI [16] uses RNN to generate API sequences for a given text by using deep learning to relate APIs. Desai *et al.* [14] synthesize domain-specific languages from English. Anycode [18] uses a probabilistic CFG with trees for Java constructs and API calls to synthesize small Java expressions. T2API [36] uses graph-based API synthesis algorithm that generates a graph representing an API usage from a large corpus.

10 CONCLUSION

In this work, we conjecture that we need a new model that fits with API documentation better than Word2vec. We present D2Vec, a neural network model that considers two complementary contexts to better capture semantics of API descriptions. First, we connect the global context of the current API topic under description to all the text phrases within the description of that API. Second, the local orders of words and API elements in the text phrases are maintained in computing the vector representations for the APIs.

We demonstrate the usefulness and good performance of D2Vec in three applications: API code search (text-to-code retrieval), tutorial fragment search (code-to-text retrieval), and mining API mappings between JDK and Apache (code-to-code retrieval).

Finally, inspired by D2Vec's success in those applications, we provide actionable insights on how to use D2Vec in the context of other SE documents including source code and in other tasks, *e.g.*, building a representation for a code fragment with an addition of a topic, applying D2Vec on bug localization with LDA, and building database of vector representations for APIs as building blocks.

ACKNOWLEDGMENTS

This work was supported in part by the US National Science Foundation (NSF) grants CCF-1723215, CCF-1723432, TWC-1723198, CCF-1518897, and CNS-1513263.

REFERENCES

- [1] M. Allamanis, E. T. Barr, C. Bird, and C. Sutton. Learning natural coding conventions. In *Proceedings of the 2014 International Symposium on Foundations of Software Engineering*, FSE'14, pages 281–293. ACM, 2014.
- [2] M. Allamanis, E. T. Barr, C. Bird, and C. Sutton. Suggesting accurate method and class names. In *Proceedings of the 10th Joint Meeting on Foundations of Software Engineering*, ESEC/FSE 2015, pages 38–49. ACM, 2015.
- [3] M. Allamanis and C. Sutton. Mining source code repositories at massive scale using language modeling. In *Proceedings of the 10th IEEE Working Conference on Mining Software Repositories (MSR'13)*, pages 207–216. IEEE CS, 2013.
- [4] M. Allamanis, D. Tarlow, A. Gordon, and Y. Wei. Bimodal modelling of source code and natural language. In *Proceedings of the 32nd International Conference on Machine Learning*, ICML '15. ACM, 2015.
- [5] J. Anvik, L. Hiew, and G. C. Murphy. Who should fix this bug? In *Proceedings of International Conference on Software Engineering*, ICSE '06, pages 361–370. ACM, 2006.
- [6] Apache documentation. <https://httpd.apache.org/docs/>.
- [7] E. Arisoy, T. N. Sainath, B. Kingsbury, and B. Ramabhadran. Deep neural network language models. In *Proceedings of the NAACL-HLT 2012 Workshop: Will We Ever Really Replace the N-gram Model? On the Future of Language Modeling for HLT*, WLM '12, pages 20–28. Association for Computational Linguistics, 2012.
- [8] S. Bajracharya, T. Ngo, E. Linstead, Y. Dou, P. Rigor, P. Baldi, and C. Lopes. Sourcerer: A search engine for open source code supporting structure-based search. In *Proceedings of the 2006 ACM International Conference on Object-oriented Programming Systems, Languages, and Applications*, OOPSLA '06, pages 681–682. ACM, 2006.
- [9] D. M. Blei, A. Y. Ng, and M. I. Jordan. Latent dirichlet allocation. *J. Mach. Learn. Res.*, 3:993–1022, Mar. 2003.
- [10] P. F. Brown, V. J. D. Pietra, S. A. D. Pietra, and R. L. Mercer. The mathematics of statistical machine translation: parameter estimation. *Comput. Linguist.*, 19(2):263–311, June 1993.
- [11] W.-K. Chan, H. Cheng, and D. Lo. Searching Connected API Subgraph via Text Phrases. In *Proceedings of the 20th International Symposium on the Foundations of Software Engineering*, FSE '12, pages 10:1–10:11. ACM, 2012.
- [12] J. Cleland-Huang, O. C. Z. Gotel, J. Huffman Hayes, P. Mäder, and A. Zisman. Software traceability: Trends and future directions. In *Proceedings of the Future of Software Engineering workshop*, FOSE'14, pages 55–69. ACM, 2014.
- [13] B. Dagenais and M. P. Robillard. Recovering traceability links between an API and its learning resources. In *Proceedings of the 34th International Conference on Software Engineering*, ICSE '12, pages 47–57. IEEE Press, 2012.
- [14] A. Desai, S. Gulwani, V. Hingorani, N. Jain, A. Karkare, M. Marron, S. R. and S. Roy. Program synthesis using natural language. In *Proceedings of the 38th International Conference on Software Engineering*, ICSE '16, pages 345–356. ACM, 2016.
- [15] A. Gokhale, V. Ganapathy, and Y. Padmanaban. Inferring likely mappings between APIs. In *Proceedings of the 35th International Conference on Software Engineering*, ICSE '13, pages 82–91. IEEE, 2013.
- [16] X. Gu, H. Zhang, D. Zhang, and S. Kim. Deep API Learning. In *Proceedings of the 2016 ACM SIGSOFT International Symposium on Foundations of Software Engineering*, FSE 2016. ACM, 2016.
- [17] L. Guerrouj, D. Bourque, and P. C. Rigby. Leveraging informal documentation to summarize classes and methods in context. In *Proceedings of the 37th IEEE/ACM International Conference on Software Engineering*, ICSE 2015, Volume 2, pages 639–642. IEEE CS, 2015.
- [18] T. Gvero and V. Kuncak. Synthesizing Java expressions from free-form queries. In *Proceedings of the 2015 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications*, OOPSLA 2015, pages 416–432. ACM, 2015.
- [19] A. Hindle, E. T. Barr, Z. Su, M. Gabel, and P. Devanbu. On the naturalness of software. In *Proceedings of the 34th International Conference on Software Engineering*, ICSE 2012, pages 837–847. IEEE Press, 2012.
- [20] K. Inoue, R. Yokomori, H. Fujiwara, T. Yamamoto, M. Matsushita, and S. Kusumoto. Component rank: Relative significance rank for software component search. In *Proceedings of the 25th International Conference on Software Engineering*, ICSE '03, pages 14–24. IEEE, 2003.
- [21] Java platform standard edition 7 documentation. <http://docs.oracle.com/javase/7/docs/>.
- [22] H. Jiang, J. Zhang, X. Li, Z. Ren, and D. Lo. A more accurate model for finding tutorial segments explaining APIs. In *2016 IEEE 23rd International Conference on Software Analysis, Evolution, and Reengineering (SANER)*, volume 1, pages 157–167, March 2016.
- [23] H. Jiang, J. Zhang, Z. Ren, and T. Zhang. An unsupervised approach for discovering relevant tutorial fragments for APIs. In *Proceedings of the 39th International Conference on Software Engineering*, ICSE '17, pages 38–48. IEEE Press, 2017.
- [24] I. Jolliffe. *Principal component analysis*. Springer Verlag, New York, 2002.
- [25] Kode java. <https://kodejava.org/>.
- [26] Q. Le and T. Mikolov. Distributed representations of sentences and documents. In *Proceedings of the 31st International Conference on Machine Learning*, volume 32 of *Proceedings of Machine Learning Research*, pages 1188–1196, Beijing, China, 22–24 Jun 2014. PMLR.
- [27] Q. V. Le and T. Mikolov. Distributed representations of sentences and documents. *CoRR*, abs/1405.4053, 2014.
- [28] C. J. Maddison and D. Tarlow. Structured generative models of natural source code. In *The 31st International Conference on Machine Learning (ICML)*, June 2014.
- [29] C. McMillan, D. Poshyanyk, and M. Grechanik. Recommending source code examples via API call usages and documentation. In *Proceedings of the 2nd International Workshop on Recommendation Systems for Software Engineering*, RSSE '10, pages 21–25. ACM, 2010.
- [30] S. Meng, X. Wang, L. Zhang, and H. Mei. A history-based matching approach to identification of framework evolution. In *Proceedings of the 34th International Conference on Software Engineering*, ICSE '12, pages 353–363. IEEE, 2012.
- [31] T. Mikolov, A. Deoras, D. Povey, L. Burget, and J. Cernocky. Strategies for training large scale neural network language models. In *Proceedings of Automatic Speech Recognition and Understanding Workshop*, ASRU'11. IEEE, 2011.
- [32] T. Mikolov, M. Karafiat, L. Burget, J. Cernocky, and S. Khudanpur. Recurrent neural network based language model. In *Proceedings of International Conference on Acoustics Speech and Signal Processing (ICASSP)*, ICASSP'10, pages 1045–1048. IEEE, 2010.
- [33] T. Mikolov, I. Sutskever, K. Chen, G. S. Corrado, and J. Dean. Distributed representations of words and phrases and their compositionality. In *27th Annual Conference on Neural Information Processing Systems 2013 (NIPS'13)*, pages 3111–3119, 2013.
- [34] L. Mou, G. Li, Z. Jin, L. Zhang, and T. Wang. TBCNN: A tree-based convolutional neural network for programming language processing. *CoRR*, abs/1409.5718, 2014.
- [35] A. T. Nguyen, H. A. Nguyen, T. T. Nguyen, and T. N. Nguyen. Statistical learning approach for mining API usage mappings for code migration. In *Proceedings of the 29th ACM/IEEE International Conference on Automated Software Engineering*, ASE '14, pages 457–468. ACM, 2014.
- [36] A. T. Nguyen, P. C. Rigby, T. Nguyen, D. Palani, M. Karanfil, and T. N. Nguyen. Statistical translation of English texts to API code templates. In *Proceedings of the 2018 IEEE International Conference on Software Maintenance and Evolution*, ICSME '18. IEEE, 2018.
- [37] T. D. Nguyen, A. T. Nguyen, and T. N. Nguyen. Mapping API elements for code migration with vector representations. In *Proceedings of the 38th International Conference on Software Engineering Companion*, ICSE '16, pages 756–758. ACM, 2016.
- [38] T. D. Nguyen, A. T. Nguyen, H. D. Phan, and T. N. Nguyen. Exploring API embedding for API usages and applications. In *Proceedings of the 39th International Conference on Software Engineering*, ICSE '17, pages 438–449. IEEE Press, 2017.
- [39] T. T. Nguyen, A. T. Nguyen, H. A. Nguyen, and T. N. Nguyen. A statistical semantic language model for source code. In *Proceedings of the 9th Joint Meeting on Foundations of Software Engineering*, ESEC/FSE 2013, pages 532–542. ACM, 2013.
- [40] M. Nita and D. Notkin. Using twinning to adapt programs to alternative APIs. In *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering - Volume 1*, ICSE '10, pages 205–214. ACM, 2010.
- [41] G. Petrosyan, M. P. Robillard, and R. De Mori. Discovering information explaining API types using text classification. In *Proceedings of the 37th International Conference on Software Engineering - Volume 1*, ICSE '15, pages 869–879. IEEE Press, 2015.
- [42] H. Phan, H. A. Nguyen, N. M. Tran, L. H. Truong, A. T. Nguyen, and T. N. Nguyen. Statistical learning of api fully qualified names in code snippets of online forums. In *Proceedings of the 40th International Conference on Software Engineering*, ICSE '18, pages 632–642. ACM, 2018.
- [43] D. Pappin and F. Silvestri. The social network of Java classes. In *SAC'06*, pages 1409–1413. ACM, 2006.
- [44] M. Raghothaman, Y. Wei, and Y. Hamadi. SWIM: synthesizing what I mean. In *Proceedings of the 38th International Conference on Software Engineering*, ICSE'16. ACM Press, 2016.
- [45] V. Raychev, M. Vechev, and E. Yahav. Code completion with statistical language models. In *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '14, pages 419–428. ACM, 2014.
- [46] P. C. Rigby and M. P. Robillard. Discovering essential code elements in informal documentation. In *Proceedings of the 2013 International Conference on Software Engineering*, ICSE '13, pages 832–841. IEEE Press, 2013.
- [47] A. D. Sorbo, S. Panichella, C. A. Visaggio, M. D. Penta, G. Canfora, and H. C. Gall. Development emails content analyzer: Intention mining in developer discussions. In *Proceedings of International Conference on Automated Software Engineering*, ASE '15. IEEE, 2015.
- [48] G. Sridhara, E. Hill, D. Muppaneni, L. Pollock, and K. Vijay-Shanker. Towards automatically generating summary comments for Java methods. In *Proceedings of the IEEE/ACM International Conference on Automated Software Engineering*, ASE '10, pages 43–52. ACM, 2010.
- [49] S. Subramanian, L. Inozemtseva, and R. Holmes. Live API documentation. In *Proceedings of the 36th International Conference on Software Engineering*, ICSE

- 2014, pages 643–652. ACM, 2014.
- [50] T. V. Nguyen, A. T. Nguyen, and T. N. Nguyen. Characterizing API elements in software documentation with vector representation. In *Proceedings of the 38th International Conference on Software Engineering Companion*, ICSE '16, pages 749–751. ACM, 2016.
 - [51] W. Wu, Y.-G. Guéhéneuc, G. Antoniol, and M. Kim. Aura: A hybrid approach to identify framework evolution. In *Proceedings of the ACM/IEEE International Conference on Software Engineering*, ICSE '10, pages 325–334. ACM, 2010.
 - [52] J. Yang and L. Tan. Swordnet: Inferring semantically related words from software context. *Empirical Softw. Engg.*, 19(6):1856–1886, Dec. 2014.
 - [53] X. Ye, R. Bunescu, and C. Liu. Learning to rank relevant files for bug reports using domain knowledge. In *Proceedings of the 22Nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*, FSE 2014, pages 689–699. ACM, 2014.
 - [54] X. Ye, H. Shen, X. Ma, R. Bunescu, and C. Liu. From word embeddings to document similarities for improved information retrieval in software engineering. In *Proceedings of the 38th International Conference on Software Engineering*, ICSE '16, pages 404–415. ACM, 2016.
 - [55] W. Zheng, Q. Zhang, and M. Lyu. Cross-library API recommendation using web search engines. In *Proceedings of the 19th ACM SIGSOFT Symposium and the 13th European Conference on Foundations of Software Engineering*, ESEC/FSE '11, pages 480–483. ACM, 2011.
 - [56] H. Zhong, S. Thummalapenta, T. Xie, L. Zhang, and Q. Wang. Mining API mapping for language migration. In *Proceedings of International Conference on Software Engineering*, ICSE '10, pages 195–204. ACM, 2010.
 - [57] J. Zhou, H. Zhang, and D. Lo. Where should the bugs be fixed? - more accurate information retrieval-based bug localization based on bug reports. In *Proceedings of the 34th International Conference on Software Engineering*, ICSE '12, pages 14–24. IEEE Press, 2012.