2017 IEEE/ACM 39th International Conference on Software Engineering

Exploring API Embedding for API Usages and Applications

Trong Duc Nguyen, Anh Tuan Nguyen, Hung Dang Phan
Electrical and Computer Engineering Department
Iowa State University
Email: {trong,anhnt,hungphd}@iastate.edu

Tien N. Nguyen
Computer Science Department
University of Texas at Dallas
Email: tien.n.nguyen@utdallas.edu

***Abstract*—Word2Vec is a class of neural network models that as being trained from a large corpus of texts, they can produce for each unique word a corresponding vector in a continuous space in which linguistic contexts of words can be observed. In this work, we study the characteristics of Word2Vec vectors, called API2VEC or *API embeddings*, for the API elements within the API sequences in source code. Our empirical study shows that the close proximity of the API2VEC vectors for API elements reflects the similar usage contexts containing the surrounding APIs of those API elements. Moreover, API2VEC can capture several similar semantic relations between API elements in API usages via vector offsets. We demonstrate the usefulness of API2VEC vectors for API elements in three applications. First, we build a tool that mines the pairs of API elements that share the same usage relations among them. The other applications are in the code migration domain. We develop API2API, a tool to automatically learn the API mappings between Java and C# using a characteristic of the API2VEC vectors for API elements in the two languages: semantic relations among API elements in their usages are observed in the two vector spaces for the two languages as similar geometric arrangements among their API2VEC vectors. Our empirical evaluation shows that API2API relatively improves 22.6% and 40.1% top-1 and top-5 accuracy over a state-of-the-art mining approach for API mappings. Finally, as another application in code migration, we are able to migrate equivalent API usages from Java to C# with up to 90.6% recall and 87.2% precision.**

***Keywords*-Word2Vec; API embedding; API usages; migration**

I. Introduction

Software library plays an important role in modern software development. To access the functionality of a library, developers use Application Programming Interfaces (API elements, APIs for short), which are the *classes*, *methods*, and *fields* provided by the library's designers. A certain combination of API elements is used to achieve a programming task and is called an *API usage*. An example of a Java Development Kit (JDK) usage for reading from a file could involve the sequence of the APIs of File and Scanner, and the control unit while to iterate over the file's contents. The combination of API elements allows intricate and complex API usages, yet, in many cases, the API usages that developers write are *repetitive*. As an evidence of that, researchers have been able to mine API usage patterns, which are frequently occurring API usages, from large code corpora [1], [2]. In other words, the sequences of API elements in API usages are *natural*, *i.e.,* have high *regularity*.

Existing works have explored the regularity of API usages to build API recommendation engines by using the statistical modeling of natural utterances and applying it to API sequences to suggest the next API call in a program editor. Typical natural language processing (NLP) models for API call suggestion include n-gram model [3], [4], deep neural network model [5], and graph-based generative model [6].

In this work, we focus on exploring the **naturalness of API usage sequences** from a different perspective by investigating the embeddings of API elements in a continuous vector space created by Word2Vec [7]. Let us call them *API embeddings* or API2VEC. Word2Vec has been shown to be able to capture the similarities of *the relations between pairs of words* in sentences: pairs of words sharing a particular relation have Word2Vec vectors with constant/similar *vector offsets*. Via visualization with Principal Component Analysis (PCA) [8] and vector operations, researchers have observed the syntactic relations, *e.g.,* (base, comparative), (base, superlative), (singular, plural), (base, past tense), etc. [9]. Semantic relations among words can also be captured via vector operations [7]. For example, for (state, capital): $V(France)$-$V(Paris) \approx V(Italy)$-$V(Rome)$, where V denotes Word2Vec and the minus sign is for vector subtraction. Other types of semantic relations are also observed: (city, state), (famous name, profession), (company, famous product), (team, sport), etc. [10]. If these observations hold for *the relations among API elements*, we could leverage API2VEC to support the tasks related to API usages, *e.g.,* API code completion, usage migration, API pattern detection, etc.

Toward that goal, we conducted experiments on a large number of Java and C# projects to answer the following research questions: 1) In a vector space produced by API2VEC on API elements, do nearby vectors represent the APIs that have similar usage contexts (defined as *similar surrounding API elements* of those APIs)? 2) By vector offsets, can API2VEC reveal similar *usage relations* between API elements (defined as *co-occurring relations between API elements in API usages*)? Our empirical results confirmed that close proximity of the vectors for two API elements reflects their similar usage contexts. We also showed that API2VEC can capture similar usage relations between the APIs in usages by vector offsets.

We demonstrated the usefulness of API2VEC vectors for APIs in three applications by using those characteristics. First, we built a tool to mine pairs of API elements sharing the same usage relations among them. For example, we can mine that the relation *"checking the existence of the current element be-*

1558-1225/17 $31.00 © 2017 IEEE
DOI 10.1109/ICSE.2017.47

fore retrieval" is shared between ListIterator.hasNext and ListIterator.next, and between StringTokenizer.hasMoreTokens and StringTokenizer.nextToken, despite that they have different names for similar functionality. Thus, given the pair <ListIterator.hasNext, ListIterator.next>, and StringTokenizer.hasMoreTokens, our tool can suggest StringTokenizer.nextToken using vector offsets.

The other applications are in the domain of code migration. We build API2API to automatically learn *API mappings* between Java and C# (*i.e.,* the API elements in two languages with the same/similar functionality). It is based on a characteristic of API embeddings in two languages as follows. In the two languages, despite that the respective APIs might have different names, since if they are used to achieve the same/similar functionality, each of the API elements would have the same/similar role in its respective API usage, and the relations between two API elements would have the same/similar meaning as the relation between the corresponding API elements in the other language. For example, the relation *"checking the existence of the current element before retrieval"* exists between ListIterator.hasNext and ListIterator.next in Java as well as between the corresponding APIs IEnumerator.MoveNext and IEnumerator.Current in C#. A usage could have multiple API elements (*e.g.,* the above API usage could include List.iterator() since we first need to obtain an iterator of a list). However, the relation between two Java API elements in a usage will exist and be interpreted as the same meaning as the relation between the two respective APIs in C#.

Thanks to the above vector offset characteristic in both Java and C# vector spaces, in our experiment, we were able to observe that the API elements in the corresponding API usages in Java and C# have *their vectors in similar* geometric arrangements *in the two vector spaces for Java and C#*. For example, Fig. 4 shows similar arrangements of the vectors for the APIs in the usages with FileReader and FileWriter in Java and StreamReader and StreamWriter in C#. This is reasonable since each element and its corresponding API element play the same/-similar role in the corresponding usage. Then, due to similar geometric arrangements, we can learn the *transformation (*e.g., *rotating and/or scaling) between the vector spaces* if we know some API mapping pairs in a training data. To find the mapped C# API for a given Java API j with its vector V_j, we use the learned function to compute its transformed vector V_C in the C# space. The C# API c with the vector most similar to the transformed vector V_C is considered as the mapped API of j.

To evaluate API2API, we conducted an empirical study. Our result shows that for a given Java API, in 53.1% of the cases, the correct respective C# API is listed on the top of API2API's resulting list. It has 22.6% and 40.1% relative improvement in top-1 and top-5 accuracy, respectively, over the state-of-the-art approach StaMiner [11]. To show the third application, we used our resulting API mappings in a phrase-based machine translation tool, Phrasal [12], to *translate a JDK usage sequence into the equivalent .NET usage with multiple API elements*. The result showed that with the mappings, our tool achieves high precision (up to 87.2%) and recall (up to 90.6%) in migrating API usages. Our key contributions include

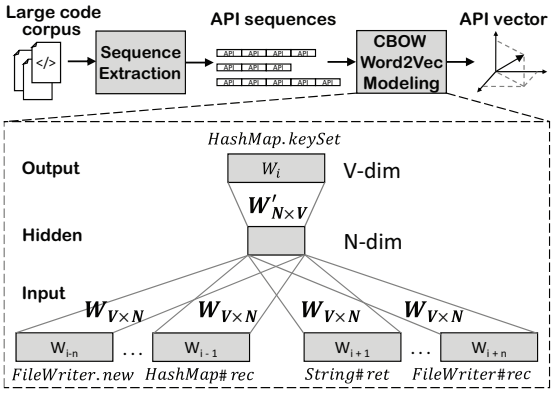

Fig. 1: Vector Representations for API Elements with CBOW

- An extensive study on the characteristics of API2VEC embeddings/vectors for API elements in API sequences;
- An application of API2VEC for API elements in mining the pairs of API elements that share same usage relations;
- API2API: an approach to mine API mappings via vector projection without a parallel corpus of respective code;
- An empirical study to show API2API's accuracy in mining API mappings, and migrating API usages from Java to C#.

II. Background on Word2Vec

Word2Vec [7] is a class of neural network models in which after being trained in a large corpus of texts, they can produce for each unique word a corresponding vector in a continuous space in which linguistic contexts of words can be observed. It represents words by encoding the contexts of their surrounding words into vectors. Mikolov *et al.* [7] introduce two Word2Vec models, named Continuous Bag-of-Words (CBOW) and Skip-gram models. We show CBOW model in Fig. 1 as we used it.

Let us summarize the CBOW model. Basically, CBOW has a neural network architecture with three layers: input, hidden, and output. The input layer has a window of n words preceding the current word w_i and a window of n words succeeding w_i. The total *(context) window*'s size is $2n$. The output layer is for w_i. Each word is encoded into the model as its index vector. An index vector for a word is an $1\times V$ vector with V being the vocabulary's size, and only the index of that word is 1 and the other positions of the index vector are zeros. The Word2Vec vector for each word w_i is the *output of the hidden layer* with N *dimensions*, which is the number of the dimensions of the vector space. To compute Word2Vec vector for w_i, CBOW first takes the average of the vectors of the $2n$ input context words, and computes the product of the average vector and the *input-to-hidden weight matrix* $W_{V\times N}$ (shared for all words):

$$V(w_i) = \frac{1}{2n}(w_{(i-n)}+...w_{(i-1)}+w_{(i+1)}+...+w_{(i+n)}).W_{V\times N}$$

$V(w_i)$ is the Word2Vec vector for w_i. $2n$ is the window's size. $W_{V\times N}$ is the input-to-hidden weight matrix. $w_{(i-n)},...,w_{(i+n)}$ are the vectors of the words in the context window. Training criterion is to derive the input-to-hidden weight matrix $W_{V\times N}$ and the hidden-to-output weight matrix

$W'_{N\times V}$ such that Word2Vec correctly classifies the current word $w = w_i$ for all words. Details can be found in [7].

III. API2VEC: API EMBEDDINGS FOR API USAGES

A. API2VEC *for API Usages*

In API usages, one needs to use API elements in certain orders. Thus, APIs are often repeatedly used in similar contexts, *i.e.,* similar surrounding API elements in which each of them has a specific role. We aim to verify if Word2Vec vectors can capture the regularity of APIs via maximizing the likelihood of observing an API element given its surrounding elements in the API usages. Let us call them **API2VEC** vectors.

Specifically, in Word2Vec, the regularity of words is expressed in two key characteristics. First, it has been shown that in the Word2Vec vector space for texts, the nearby vectors are the projected locations of the words that have been used in the similar contexts consisting of similar surrounding words [13]. Thus, our first research question is **to verify whether the close proximity of the vectors in the API2VEC vector space represent the API elements that have similar usage contexts**. *Two APIs have similar usage contexts if they have similar sets of surrounding API elements in their API usages*. Examples of APIs with similar contexts are the APIs in the same class or the classes with similar purposes (*e.g.,*StringBuffer and StringBuilder). They are often surrounded by similar sets of APIs in usages.

Second, in NLP, the regularity of words is observed as similar vector offsets between the pairs of words sharing a particular relation. For API usages, APIs are used in certain ways with semantic dependencies/relations among them. For example, the relation *"check if the current element exists before retrieval"* occurs between ListIterator.hasNext and ListIterator.next, and between XMLStreamReader.isEndElement and XMLStreamReader.next. Such relations among APIs, called *usage relations*, are parts of API usages and occur regularly in source code. Thus, our second research question is **to verify if similar usage relations between APIs can be observed via vector offsets**.

B. Building API Sequences for API Usages

For training, we process a large code corpus to build the sequences of *annotations* to represent API elements in usages. We traverse an AST to build an *annotation sequence* according to the syntactic units related to APIs, including *literals*, *identifiers*, *API elements* (*method/constructor calls*, *field accesses*), *variable declarations*, *array accesses*, and *control statements* (while, for, if, etc.). For a non-control unit, we collect data types, method/field names, return types, and roles (literals, variables, receivers/arguments). Such annotation is expected to increase regularity and characterize API elements. The names of types/-classes/methods/fields are kept. Those of variables/identifiers are discarded since different usages could use different names.

Table I shows the key rules to build annotation sequences in Java. Similar rules are used for C# (not shown). θ is used to denote the function to build an annotation sequence (*API sequence* for short). It is initially applied on a method and recursively called upon the syntactic units in the code until we have all terminal annotations. A *terminal annotation is either a method call* (e.g.,*Reader.read), field access, or a type with/without a suffix annotation* (*e.g.,*String, FileWrite#var, HashMap#rec, String#arg). The final sequence contains only terminal ones.

TABLE I
KEY RULES $\theta(E)$ TO BUILD API SEQUENCES IN JAVA

Syntax	T = typeof, RetType = return type
Expression	
Literal: E ::= Lit	$\theta(E)$ = T(Lit) *e.g.,* θ("ABC") = String
Identifier E ::= ID	$\theta(E)$ = T(ID)#var *e.g.,* θ(writer)=FileWriter#var
MethodCall E ::= e.m(e_1,...,e_n)	$\theta(E) = \theta(e_1)$ ... $\theta(e_n)$ RetType(m)#ret $\theta(e)$#rec T(e).m T(e_1)#arg ... T(e_n)#arg Discard $\theta(e_i)$ if e_i is ID or Literal Discard θ(e)#rec if e is a class name *e.g.,* θ(dict.get(vocab)) = Integer#ret HashMap#rec HashMap.get String#arg
Constructor E ::= [e.]new C(e_1,...,e_n)	$\theta(E) = \theta(e_1)$ ... $\theta(e_n)$ [$\theta(e)$] T(C).new T(e_1)#arg...T(e_n)#arg *e.g.,* θ(new FileWriter("A")) = FileWriter.new String#arg
Field Access E::= e.f	$\theta(E)$ = T(f)#ret θ(e)#rec T(e).f Discard $\theta(e)$#rec if e is a class name *e.g.,* θ(reader.lock)=Object#ret Reader#rec Reader.lock
Variable Decl E ::= C id_1[=e_1], ...[id_n[=e_n]]	$\theta(E)$ = C#var $\theta(e_1)$ [... C#var $\theta(e_n)$] *e.g.,* θ(FileWriter writer)=FileWriter#var
ArrayAccess E::= a [e]	$\theta(E) = \theta$(e) T(a[]) T(a)#access T(e)#arg Discard $\theta(e)$ if e is ID or Literal *e.g.,* θ(list[1]) =String String[]#access Integer#arg
Lambda expr E ::=(e_1,...,e_n) => e	$\theta(E) = \theta(e_1)...\theta(e_n)$ T(e_1)#arg ... T(e_n)#arg $\theta(e)$
Statement	
ForStmt S::= for (i_1,...,i_n ; e; u_1, ..., u_m) S1	$\theta(S)$='for' $\theta(i_1)$ ... $\theta(i_n)$ $\theta(e)$ $\theta(u_1)$ ... $\theta(u_m)$ θ(S1) *e.g.,* θ(for (; it.hasNext();)) = for bool Iterator#var Iterator.hasNext
S::= while (e) S1	θ = 'while' $\theta(e)$ θ(S1)
S::= if (e) S1 [else S2]	θ(S) = 'if' $\theta(e)$ θ(S1) 'else' [θ(S2)]
ExprStmt S::= e ;	θ(S) = θ(e)
Block S::=s_1,..,s_n	θ(S) = $\theta(s_1)$ ... $\theta(s_n)$

1) Literal: We keep only its type.
2) Identifier: We concatenate its type with annotation #var.
3) Method call: We keep its full signature including the return type and the types (not the concrete names) of its receiver and arguments. For example, for dict.get(vocab), we have Integer#ret HashMap#rec HashMap.get String#arg. Such type information could help predict the current API call given the return type and its arguments' types, or predict the current argument given the name of API call, its return type, and other arguments. We keep the receiver's type since we want to capture the following relations: an object "invokes" an API call, and a call "returns" an object with a specific type. If a method call is an argument of another call, $m(n())$, the sequence for $n()$ is created before the one for $m()$ because $n()$ is executed first.
4) Constructor call or field access: similar to method call. For a constructor call, no return type and receiver's type is needed.
5) Variable declaration: We keep its type and annotation #var, *e.g.,*FileWriter#var. *We discard its name* to increase its regularity.
6) Array access: We keep the types of the array, the elements, and the index, *e.g.,*list[1] → String String[]#access Integer#arg.
7) Statements: The rules for while, for, if, and other statements are similar, however, those keywords are also kept.

```
1 HashMap dict = new HashMap();
2 dict.put("A", 1);
3 FileWriter writer = new FileWriter("Vocabulary.txt");
4 for (String vocab: dict.keySet())
5     writer.append(vocab + " " + dict.get(vocab)+"\r\n");
6 writer.close();
```

Fig. 2: An API Usage in Java JDK

TABLE II
DATASETS TO BUILD API2VEC VECTORS

	#projects	#Classes	#Meths	#LOCs	Voc size
Java Dataset	14,807	2.1M	7M	352M	123K
C# Dataset	7,724	900K	2.3M	292M	130K

For example, in Fig. 2, we produce the API sequence:
HashMap#var HashMap.new
String#ret HashMap#rec HashMap.put String#arg Integer#arg
FileWriter#var FileWriter.new String#arg
for String#var String[]#ret HashMap#rec HashMap.keySet
String#ret HashMap#rec HashMap.get String#arg FileWriter#rec
FileWriter.append String#arg
FileWriter#rec FileWriter.close

Such sequences for all methods in a dataset is used to train CBOW. An API (annotation) sequence for a method is called *a sentence*. Each element in a sentence is considered as the current one. Assume that the current API is HashMap.keySet (used for the output layer, Fig. 1). If the context window's size is 10, we use 5 elements preceding and 5 succeeding it for the input. After training, the output of the hidden layer gives the vector for the current API, which is called **API embedding**.

IV. CHARACTERISTICS OF API EMBEDDINGS

We conducted experiments to answer the following questions:

RQ1. In a vector space for the APIs in usages, do nearby vectors represent the APIs that have similar usage contexts?

RQ2. Can vector offsets in API2VEC capture similar usage relations (*i.e.,co-occurring relations* among APIs in usages)?

Data Collection (Table II) The first dataset, collected from Allamanis *et al.* [14], is for training Word2Vec model to build the vectors for JDK APIs. For vectors for .NET APIs, we chose 7,724 C# projects with the ratings of +10 stars in GitHub.

A. RQ1. Nearby Vectors Represent APIs with Similar Contexts

We first randomly selected 1,000 JDK API methods and fields in our dataset. For each API, we computed the top-5 API method calls and field accesses that are closest to that API in the vector space. We processed those 1,000 groups of 6 API methods/fields (one main API of the group and top-5 closest ones) to verify if each of those 5 elements shares similar usage contexts with the main API (*i.e.,* used with similar surrounding APIs). For such verification, we wrote a program to take two APIs a and b and search through our Java dataset to compute two sets A and B of API elements that have been frequently occurred with a and b, respectively (80% threshold), in the methods in the dataset. If A and B overlaps more than 80%, we consider a and b share similar surrounding APIs in usages.

TABLE III
EXAMPLES OF APIS SHARING SIMILAR SURROUNDING APIS

G1. File.new	**G4. List.iterator**
System.getProperty	SynchronousQueue.iterator
ProcessBuilder.directory	ArrayList.iterator
Path.toFile	ArrayDeque.iterator
FileDialog.getFile	Collection.iterator
JarFile.new	Vector.iterator
G2. System.currentTimeMillis	**G5. String.hashCode**
Calendar.getTimeInMillis	Integer.hashCode
ThreadMXBean.getThreadUserTime	Date.hashCode
Thread.sleep	Class.hashCode
File.setLastModified	Boolean.hashCode
Calendar.setTimeInMillis	Long.hashCode
G3. String.compareTo	**G6. Map.keySet**
Integer.compareTo	IdentityHashMap.entrySet
Comparable.getClass	EnumMap.entrySet
Boolean.compareTo	AbstractMap.keySet
Long.compareTo	NavigableMap.keySet
Comparable.toString	IdentityHashMap.keySet

TABLE IV
t-TEST RESULTS FOR VECTOR DISTANCES OF APIS IN THE SAME AND DIFFFERENT CLASSES AND PACKAGES FOR JAVA AND C#

	t	df	p-value	Confidence interval
Java Class	-934.33	223.330	$<2.2\text{x}10^{-15}$	(-∞; -0.5280486)
Java Package	-109.52	67.360	$<2.2\text{x}10^{-15}$	(-∞; -0.0472560)
C# Class	-962.47	351.961	$<2.2\text{x}10^{-15}$	(-∞; -0.6252377)
C# Package	-443.71	282.878	$<2.2\text{x}10^{-15}$	(-∞; -0.1364794)

Among 5,000 pairs of APIs (1,000 groups and 5 comparisons each), we found that 4,632 pairs (92.64% of them) have similar surrounding APIs in their usages. Thus, this gives a positive answer to our RQ1. For the other 7.36%, this is because an API has multiple contexts and some contexts with infrequently used APIs were not captured with insufficient data.

Table III displays a few groups of those nearby API vectors. The 3 groups on the left share similar surrounding API elements despite that their names are quite different. The 3 groups on the right have members sharing the names. For illustration, we show only the groups with the members in different classes.

B. Vectors of the APIs in Same Classes/Packages

In this experiment, we aim to verify if *an API method call or field access to be projected closer to the other APIs of the same class than the APIs of different classes* (*).

We computed the cosine distances among the vectors of the API methods and public fields in the same class and the distances among the vectors of the APIs from different classes. For every API method/field m, we computed the distances from m to all other API method/fields in the same class with m and to all other methods/fields in different classes. To verify (*), for all the distances in the entire set of APIs, we conducted the independent-samples t-test with significance level $\alpha = 0.99$. We chose the following alternative hypothesis: "*the distances among the vectors of the APIs within a class are smaller than the distances among the vectors of APIs belong to different classes*". The null hypothesis is *"those distances are equal"*. We also performed the same procedure for the

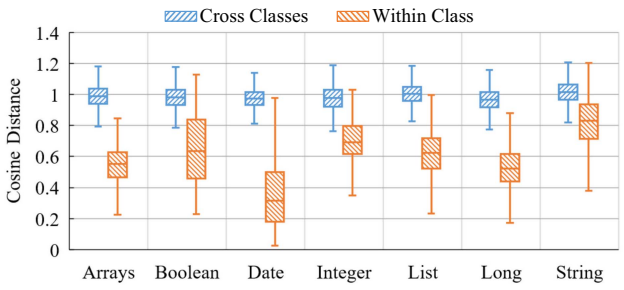

Fig. 3: Distances of JDK API Vectors within and cross Classes

API methods/fields with respect to the boundary of packages. Table IV shows the results for both Java and C# vectors. As seen, with the p-values, we can confirm our alternative hypothesis: the distances among the vectors for APIs in the same class/package is significantly smaller than the distances for APIs in different classes/packages. Thus, we can verify (*).

Fig. 3 shows the boxplot for the distributions of distances among the vectors of the methods/fields in the same classes for the 7 most popular JDK classes in our dataset. We also show the boxplot for the distributions of distances between the vectors of the APIs in each class and those in other classes. As seen, the two boxplots for each class are quite separated.

In brief, *the APIs in the same class/package perform functions relevant to the class/package's theme, and often share similar surrounding APIs. They tend to have nearby vectors*.

C. RQ2. Similar Vector Offsets Reflect Similar Relations

We first mined the frequent pairs of APIs by collecting all the pairs of the APIs in the same methods in our Java dataset. We ranked the pairs by their occurrence frequencies. We then manually checked the most frequent pairs and collected 120 of them, which are placed into 14 groups of pairs representing 14 different relations. Similarly, we collected a set of 138 correct pairs of C# APIs placed into 16 groups. We used those two sets of pairs in JDK and .NET as our benchmarks.

We processed the pairs as follows. For each group of pairs of APIs (representing a relation), we randomly picked a seed pair, *e.g.,* (List#var, List.add). For each of the other pairs in the group, *e.g.,* (Map#var, Map.put), we applied the vector offset from the seed pair to the vector of the first API of the current pair to compute the resulting vector, *e.g.,* $X = V(\text{List.add}) - V(\text{List\#var}) + V(\text{Map\#var})$. We then searched for the vectors that are closest to X (*e.g.,*Map.put) and considered them as the candidates (ranked by their respective cosine distances). If the second API of the current pair in the benchmark is in the top-k of the candidate list, we count it as a hit. Accuracy is the ratio between the number of hits over the total number of cases.

There are 94.2% of the correct APIs in those relations showing up in the top-5 candidate lists. 74.1% are actually at the top one. Table V shows examples of 5 groups of relations in our oracle for JDK APIs and the ranks of the correct APIs in the candidate lists. As seen, API2VEC can capture similar relations between APIs and rank highly the correct APIs, even when the respective names are different. For example, in the relation

TABLE V
EXAMPLE RELATIONS VIA VECTOR OFFSETS IN JDK

R1. Check the current element before retrieval		Rank
ListIterator.hasNext	ListIterator.next	1
Enumeration.hasMoreElements	Enumeration.nextElement	1
StringTokenizer.hasMoreTokens	StringTokenizer.nextToken	3
XMLStreamReader.isEndElement	XMLStreamReader.next	1
R2. Obtain property after creating system/stream		
System#var	System.getProperty	1
Properties#var	Properties.getProperty	1
XMLStreamReader#var	XML...Reader.getAttr...Value	1
R3. Add an element to various types of collections		
List#var	List.add	1
Map#var	Map.put	1
Hashtable#var	Hashtable.put	1
Dictionary#var	Dictionary.put	1
R4. Parse a string into different types of numbers		
Float#var	Float.parseFloat	1
Double#var	Double.parseDouble	1
Integer#var	Integer.parseInt	1
Long#var	Long.parseLong	1
R5. Avoid adding duplicate element to a collection		
Set.contains	Set.add	1
Map.containsKey	Map.put	3
LinkedList.contains	LinkedList.add	1
Hashtable.containsKey	Hashtable.put	3

"add an element to various types of collections", as using List, one uses List.add, however, Map.put is used for Map. We were also able to observe/interpret the same relations in C#:

- *"Check size before removal"*, *e.g.,*Dictionary.Count − Dictionary.Remove,
- *"Add an element to a collection"*, *e.g.,*Hashtable.new − Hashtable.Add,
- *"Read a file with different types"*, *e.g.,*BinaryReader.ReadInt64 − System.Int64,
- *"Check the current element before retrieval"*, *e.g.,*IEnumerator.MoveNext − IEnumerator.Current, etc.

We also build a tool to derive pairs of API elements with the same/similar relations. The tool takes as input a pair of API elements in the same class, e.g., List#var and List.add, and another API in a different class, e.g., Hashtable#var. It then uses the vector offsetting operation to derive the corresponding API Hashtable.put, without understanding the meaning of the relation. One could use this tool to derive that List.add could be used to achieve the similar functionality as Hashtable.put. This is useful for developers who are new to the APIs.

V. MINING API MAPPINGS BETWEEN JAVA AND C#

A. API Mappings

This section presents an application of API embeddings in code migration. Migrating code from one language to another requires not only the mappings between the language constructs (*e.g.,* statements, expressions), but also the mappings between the APIs in the two languages that have the same/similar functionality. For example, in JDK, one uses System.out.println, while in .NET, (s)he could use Console.WriteLn. To

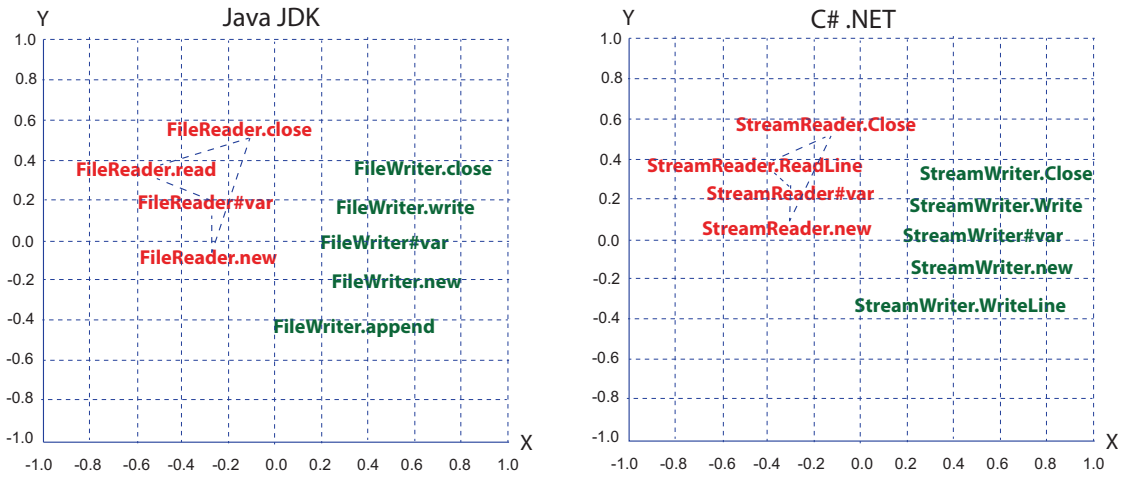

Fig. 4: Distributed Vector Representations for some APIs in Java (left) the corresponding APIs in C# (right)

reduce manual effort, several approaches have been proposed to *mine API mappings* from a parallel training corpus of the libraries' client code that already had two respective versions in two languages [15], [16]. However, building such corpus with parallel implementations in general requires much effort.

B. Mappings via Transformation between Two Vector Spaces

In this work, we introduce API2API, an approach/tool to automatically mine API mappings between Java and C# without requiring a parallel corpus. API2API is based on a characteristic of the API2VEC vectors for API elements in two languages: semantic relations among APIs in their usages are observed in the two vector spaces for the two languages as *similar geometric arrangements among their vectors*.

For motivation, we conducted an experiment in which we picked 2 groups of APIs in Java JDK, FileReader and FileWriter, and the corresponding APIs in C# .NET. The vectors for the respective APIs in JDK and .NET in each group were projected down to two dimensions using PCA [8] (Fig. 4). We visually observe that the group of FileReader and that of the respective one StreamReader have similar geometric arrangements in the two vector spaces. This suggests further exploration. With this projection to 2-D spaces, we were able to compute a transformation matrix that converts those two groups of APIs in Java to the respective ones in C#. That is, similar geometric arrangements enable us to find a transformation with rotating and scaling between the vectors in the two vector spaces.

The rationale is that the usage relations, *e.g.,* in the usage *"open a file, read, and close it"* (among FileReader-#var, FileReader.new, FileReader.read, and FileReader.close) are observed as the vector offsets in the Java API vector space. In C#, those usage relations are also captured via the vector offsets among the corresponding APIs in the C# vector space (StreamReader#var, StreamReader.new, StreamReader.ReadLine, and StreamReader.Close). The distance (vector offset) between the API vectors with such a relation in the Java space might be different from the distance between the

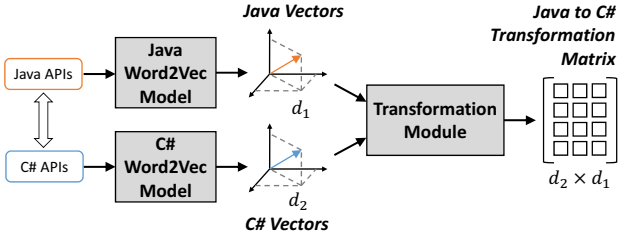

Fig. 5: Training for Transformation Matrix

corresponding API vectors with the same relation in the C# space. However, as in NLP, such a distance (vector offset) for two API vectors in Java space *can be interpreted as the same relation* as the distance (offset) between two vectors for the corresponding APIs in C# space. For example, both V(FileWriter.new)-V(FileWriter.append) and V(StreamWriter.new)-V(StreamWriter.WriteLine) can be interpreted as the relation *"open and append to a file"*. Thus, the respective vectors in two vector spaces could form similar geometric arrangements.

C. Transformation Matrix to Compute Single API Mappings

From the above observation, we aim to learn the transformation between the two vector spaces for APIs from some prior-known API pairs, and then use the learned transformation to locate the C# vectors corresponding to other unknown Java APIs. Fig. 5 shows how we train the transformation model. First, we collect the single mappings between JDK in Java and .NET in C# into a training set (in our empirical evaluation, we used a set of API mappings that was provided as part of the migration tool Java2CSharp [17]). For example, FileReader in JDK is mapped to StreamReader in .NET. Then, we use the trained Word2Vec models for JDK and .NET to collect the vectors for all the pairs of APIs in the training set.

In training, the pairs of vectors of those respective APIs are used to derive the transformation matrix from Java to C# as follows. Let us have a training set of API pairs and their associated vector representations $\{j_i, c_i\}, i = 1..n$ where j_i is a

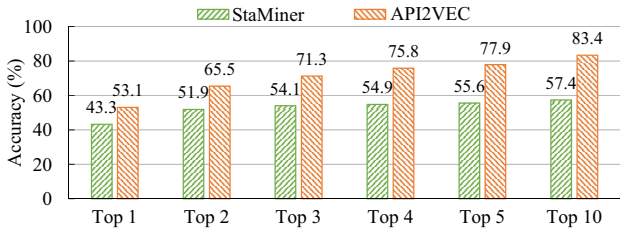

Fig. 6: Comparison in Top-k API Mapping Mining Accuracy

vector in the Java vector space with d_1 dimensions and c_i is the corresponding vector in the C# space with d_2 dimensions. We need to find a transformation matrix T such that $T \times j_i$ approximates c_i. Adapted from [18], we learn the matrix T with the dimensions $d_2 \times d_1$ by minimizing the Least Square Errors:

$$\min_{W} \sum_{i=1}^{n} ||T \times j_i - c_i||^2$$

Training process is done with stochastic gradient descent [18].

For prediction, for a given API in Java j, we compute $c = T \times j$. The API in C# whose vector is closest to c via cosine similarity will be the top result. We produce multiple candidates with their scores using the cosine similarity measures. For all JDK APIs in its vocabulary, we use the computed matrix to compute their corresponding single mappings in .NET in C#. That is, we have $\{j_i, c_i\}, i = 1..|V|$ with V is the vocabulary of JDK APIs. Note that, our mining approach works in the other direction as we can compute the transformation matrix from C# to Java in the same manner.

D. Accuracy Comparison

This section presents our experiments to evaluate API2API's accuracy in mining API mappings between Java and C#.

Beside the two datasets (Table II) to train the respective Word2Vec models in Java and C#, we also used 860 API mapping pairs between Java JDK and C# .NET, provided by the migration tool Java2CSharp [17] as the oracle. We used part of those mappings to compute the transformation matrix. For a test JDK API j, API2API produces a resulting list. If the true mapping API in C# .NET for j is in the top-k resulting list, we count it a hit. Top-k accuracy is computed as the ratio between the number of hits and the total of hits and misses.

1) **Quantitative Comparison***:* We conducted an experiment to compare API2API with the state-of-the-art approach StaMiner [11]. StaMiner could mine both single API mappings as well as the mappings for the usages with multiple API elements. For comparison with API2API, we configured StaMiner to mine single API mappings. In StaMiner [11], the authors showed that it performs better than the existing mining approaches such as MAM [15], AURA [19], and HiMa [20], thus, we do not compare with those existing tools.

We trained StaMiner in the same dataset used in its paper (Table 2 of [11]) with 34,628 pairs of respective methods in Java and C# in 9 projects that have been developed in Java and (semi-)automatically ported to C#. For API2API, we used the training datasets in Table II to produce Word2Vec vectors for the APIs. We configured API2VEC with the number of dimensions of vector spaces N=300, and the window size of Word2Vec model 2*n=10. For both tools, we used 10-fold cross validation on Java2CSharp's API mapping dataset.

TABLE VI
NEWLY FOUND API MAPPINGS, NOT IN JAVA2CSHARP BENCHMARK

Java API	C# API
java.util.HashMap.size	System.Collect...Generic.Dictionary.Count
java.util.List.size	System.Collections.Generic.IList.Count
java.util.Map.Entry.getKey	System.Coll...Generic.KeyValuePair.Key
java.util.ArrayList.ensureCapacity	System.Collections.Generic.List.Capacity
java.sql.ResultSet.getShort	System.Data...SqlDataReader.GetInt16
java.sql.ResultSet.getInt	System.Data.....SqlDataReader.GetInt32
java.sql.ResultSet.getLong	System.Data...SqlDataReader.GetInt64
java.io.File.canWrite	System.IO.FileInfo.IsReadOnly
java.io.InputStream.read	System.IO.Stream.ReadByte

As seen in Fig. 6, API2API achieves high accuracy. For a given Java JDK API, it can correctly derive the corresponding API in C# .NET in **53.1% of the time with just a single suggestion**. That is, *the correct corresponding .NET API is on the top of the resulting list in more than half of the suggestion cases*. Moreover, *for a given JDK API, the correct corresponding API in .NET is in the list of 5 suggested .NET APIs in almost 4 out of 5 cases* (**77.9%**). That is, *in 77.9% of the time, users just need to check a list of 5 suggested APIs to find the correct C# API for a given JDK API*. This result shows that it is practical to use API2API in helping code migration.

Importantly, as seen, API2API *outperforms* StaMiner about 10% at top-1 accuracy, i.e., *22.6%* relative improvement. At top-5 accuracy, the relative improvement is 40.1%.

API2API is able to detect a large number of pairs of APIs with different names. Some examples are shown in Table VI.

2) **Qualitative Comparison***:* Investigating further the result, we reported the (dis)advantages of two approaches. First, StaMiner requires a parallel corpus of corresponding usages in two languages. It is not always easy to collect a statistically significant number of parallel code. Second, both tools have out-of-vocabulary issue, *i.e.,* requiring to see the APIs in the training dataset to produce their mappings. Third, StaMiner has a stronger requirement that *the mapped APIs must be in respective pairs in the parallel corpus*. Using transformation, API2API does not need a parallel corpus with respective API usages. However, it requires a training dataset of single API pairs. It would be better if the training API pairs are diversely selected in multiple packages (discussed in Section V-E1). Fourth, it needs a high volume of code to build high-quality vectors. This issue is easily mitigated due to a large wealth of open-source repositories. In this study, with our easily-collected datasets (Table II), API2API performs better than StaMiner with 34,628 pairs of respective methods. Finally, this result leads to a potential direction to combine two approaches.

3) **Newly Found API Mappings***:* Interestingly, we found that API2API correctly detected a total of 52 new API mappings that were not manually written in the latest mapping file

Fig. 7: Top-k Accuracy with different Training Data Selections

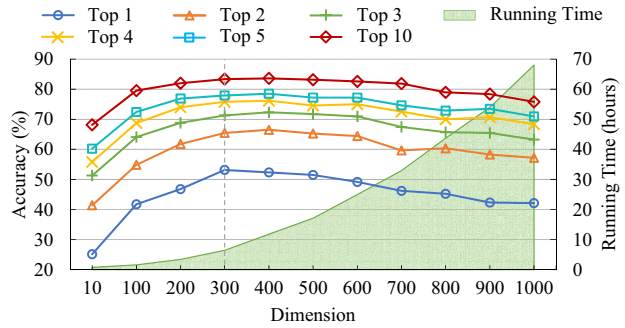

Fig. 8: Top-k Accuracy with different Numbers of Dimensions

in Java2CSharp (currently, we counted them as incorrect cases since those mappings are not in the Java2CSharp benchmark. Thus, API2API's actual accuracy is even higher). Some cases with different syntactic types and names are listed in Table VI (see our website [21] for more). Those newly found mappings are correct and could be added to complement Java2CSharp's data file. StaMiner can detect only 25 new mappings.

E. Sensitivity Analysis: Impacts of Factors on Accuracy

1) **Selecting different packages of API mapping pairs to train the transformation matrix***:* As shown in Section IV-B, the vectors for APIs in the same classes/packages are closer than those for other APIs in different classes/packages. Thus, we aimed to answer the question of whether this characteristic affects the training quality of the transformation matrix and consequently affects accuracy. We first divided our dataset of all 860 API mappings into 13 groups corresponding to 13 JDK packages. We used one group of mappings for testing, and the other 12 groups for training. We repeated the process with every group as the testing group and accumulatively measured the top-k accuracy. We compared this accuracy with that in which we conducted 10-fold cross validation with the mappings in the training set *being randomly selected from every package* (each package must have at least one pair).

As seen in Fig. 7, randomly selecting training mappings in more diverse packages gives better accuracy than the first setting. For top-1 accuracy, the difference is 53.1%-35.0%= 18.1%. In the first setting, the lack of mappings in the package used for testing really hurts accuracy. This result implies that in addition to the large size of training data, we need to have a diversity in API mappings used for training. Investigating further from the result in Section IV-B, we found that the vectors for APIs in the same classes/packages or for APIs sharing similar surrounding API elements are *clustered into groups* of nearby vectors. We found that the vectors of JDK APIs in the same cluster have similar arrangements as the corresponding vectors of .NET APIs in the respective cluster. Thus, *if we provide the mappings for some APIs in a cluster, they likely help derive other mappings in the cluster since they provide better information to learn the transformation matrix.*

There are two implications from this result. First, if we want to derive the API mappings in some package, we need to have in the training data the pairs of APIs from that package. Second, if one aims to manually build a training set of mappings, (s)he needs to diversify the pairs in JDK packages.

2) **Varying Numbers of Dimensions of Vector Spaces***:* The dimension N of vector space (Section II) is a crucial factor that could affect API2API's accuracy. In this experiment, we configured the dimensions of the two Word2Vec models for Java and C# APIs ranging from N_{java}=$N_{C\#}$=N=10, 100, 200,..., 1,000. We performed 10-fold cross validation on the pairs of API mappings. We also measured running time.

Fig. 8 shows the result. As seen, the very low-dimensional vector spaces give low accuracy, *e.g.,* 25.1% top-1 accuracy for N=10. As we increase N, accuracy increases gradually and reaches its peak (across all top-k accuracy values) around N=300. This is reasonable since the low-dimensional vector space does not fully capture the APIs' characteristics with regard to their surrounding APIs in usages. Multiple features are compressed into same dimensions. When N is large enough, the characteristics of APIs are better captured, leading to higher accuracy. However, as we increase N more (N>=400), accuracy starts to decline gradually. In this case, a more complex model with larger N requires larger training data.

As seen in Fig. 8, training time increases significantly as N>=300-400 due to the significant increase in the numbers of models' parameters. To achieve both high accuracy and reasonable training time, we use N=300 (6 hours of training) as the default configuration for subsequent experiments. Time to derive a mapping for an API is *a few milliseconds* (not shown), thus, API2API is suitable to be interactively used in an IDE.

3) **Varying Word2Vec Window's Sizes***:* We varied the size of the window 2*n of the Word2Vec model and measured its impact on accuracy. When n is small (n=1–2), the context is insufficient to represent each API element in API usages. For our dataset, with the window's size of 10–12 (5–6 APIs before and 5–6 APIs after the current API), accuracy reaches its peak (not shown) since the window can cover well the length of API sequences in an API usage. When the window's size is larger (>12), running time increases much, while accuracy is stable. Thus, we use window's size of 10 as a default setting.

4) **Varying Sizes of Training Datasets for Word2Vec***:* We varied the sizes of both training datasets in Java and C# (Table II). First, we randomly selected 2% of all the methods in Java dataset and 2% of the methods in C# dataset to train the Word2Vec models. We repeated the 10-fold cross validation as in the previous study and measured top-k accuracy. Next, we increased the training data's sizes for both Java and C# by randomly adding more methods to reach 5%, 10%, 25%,

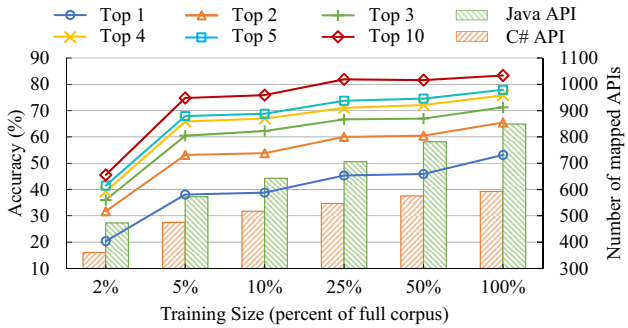

Fig. 9: Accuracy with Varied Training Datasets for Word2Vec

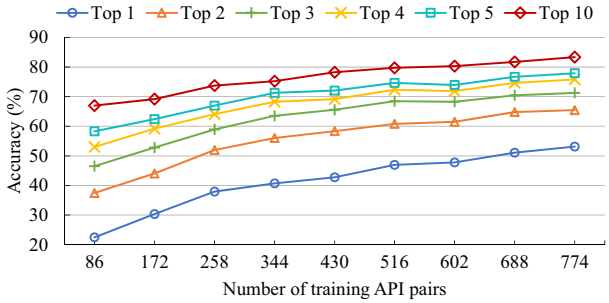

Fig. 10: Accuracy with various Numbers of Training Mappings

50%, and full training corpora. As seen in Fig. 9, as more training data added, API2API *encounters more APIs and usage contexts*, and the regularity of APIs increases. Moreover, as seen, *more mapped APIs were observed*, we trained better the transformation matrix, thus leading to higher accuracy.

5) **Varying number of mapping pairs to train the transformation matrix***:* In this experiment, we varied the size of the dataset to train the transformation matrix. We divided all 860 API mappings from Java2CSharp into 10 equal folds. First, we chose the first fold as the *testing fold*. We then used the second fold for training and measured accuracy. Next, we added the third fold to the current training data (consisting of the second fold) and tested on the testing fold. We repeated the process by adding more folds to the current training data until the 10^{th} fold was used. Then, we chose the second fold as the testing fold and repeated the above process. The top-k accuracy for each size of training data was accumulatively computed over all executions for that size. Note that, we do not need 9 folds for training to run for one fold. In fact, API2API produced mappings for all the Java APIs not in the training set.

As seen in Fig. 10, as more training mappings are added, top-k accuracy increases across all ks. Top-1 accuracy increases from 22.4% to 53.1% when training data increases from 1 to 9 folds (86 to 774 mappings). Importantly, with only 10% of data, it achieves *60% top-5 accuracy*. As 30% of the mappings (258) are used, it achieves high top-1 accuracy (40%).

VI. Migrating Equivalent API Usage Sequences

We conducted another experiment to show a useful application of our approach. We used API2API's single API mappings in a phrase-based machine translation tool, Phrasal [12], that *takes a JDK API usage and produces a respective usage sequence with multiple API elements in .NET*. For example, given the Java code in Fig. 2, our Phrasal-based tool, equipped with API2API's API mappings, will produce a sequence of APIs in C#: Dictionary#var, Dictionary.new, StreamWriter#var, etc. Users will fill out the concrete variables to produce the complete code. (We did not aim to migrate complete code since it requires the mappings of all constructs in Java and C#.)

TABLE VII
Accuracy (%) in Generating Equivalent API Usage Sequences

Project	Within-Project		Cross-Project	
	Recall	Precision	Recall	Precision
Antlr	87.8	75.2	**90.6**	**87.2**
db4o	83.9	79.4	88.7	75.8
Fpml	**89.6**	**86.1**	86.3	83.7
Itext	75.9	77.2	76.5	81.3
JGit	77.2	**66.4**	81.1	**67.1**
JTS	76.3	76.6	76.3	73.7
Lucene	**75.7**	77.7	**77.1**	78.5
Neodatis	78.6	70.4	78.8	74.2
POI	76.9	78.3	77.1	78.6

Settings and metrics. We used the dataset O of 34,628 pairs of respective methods as in the study in Section V-D1. We parsed the methods to build the API sequences, and used them and API2API's single mappings to train Phrasal.

We have two settings. The first one is within-project usage migration, which supports the case that users partially migrated a project and our tool helps in migrating the remaining methods. For each project, we used 10-fold cross validation on all of its methods. We compared the resulting sequences of APIs in C# with the real sequences in the manually-migrated C# code in the dataset O. The second setting is cross-project migration, which supports the case that developers can use our tool to migrate the usages for a new project while using the migrated usages in the other projects for training. In this setting, we used the API sequences in the methods of a project for testing and those in the remaining projects for training. We repeated the process for each of those projects, and compared the result against the human-migrated oracle O.

To measure accuracy in migrating API usages, we computed precision and recall of our translated sequences while also considering the orders of APIs. We computed the longest common subsequence (LCS) of a resulting sequence and its reference sequence in the oracle. Precision and recall values are computed as: $Precision = \frac{|LCS|}{|Result|}$, $Recall = \frac{|LCS|}{|Reference|}$. They are accumulatively computed for all resulting sequences. The higher $Recall$, the higher coverage the migrated sequences. $Recall$=1 means that the migrated sequences cover all the APIs in the oracle in the right order. The higher $Precision$, the more correct the migrated sequences. $Precision$=1 means that the migrated APIs are all correct and in the right order. We also computed BLUE score for lexical matching [11].

Result. As seen in Table VII, the results in both settings are comparable since JDK and .NET APIs are very popular. Our Phrasal-based tool is able to *migrate API sequences from Java to C# with high recall and precision*. Specifically, for a given

sequence of JDK API elements, the *first resulting .NET API sequence from our tool covers from 7.6–8.9 out of 10 needed .NET APIs, and from 6.6–8.6 out of 10 generated .NET API elements are in the correct order in that API sequence*. Thus, users just need to remove 1.4–3.4 .NET APIs and search for additional 1–2.4 APIs out of 10 elements. BLUE scores (n=4) are from 65.2–76.5%. Accuracy in the cross-project setting is slightly higher than that in the within-project one since the model observed more diverse API usages in other projects. The improvement is not much due to a small number of projects.

VII. Threats to Validity and Limitations

Our datasets and the randomly selected sets of API pairs might not be representative. For API mappings, we did not train and test of API mappings on the same package due to the small number of samples for each package in Java2CSharp oracle. For fair comparison between models, we measured in-vocabulary accuracy (counting cases with already-seen APIs).

Due to space limit, we did not report the result from Skip-gram model. CBOW predicts target words from the words in their contexts, while Skip-gram predicts context words from the target words. We focused on CBOW since it fits with our need better in characterizing an API via its surrounding elements. We tried both and CBOW gave slightly better accuracy.

API2API has shortcomings. First, it works best with one-to-one mappings. It cannot handle the cases with n-to-1 or 1-to-n mappings, and the case of mappings to multiple alternative subclasses of a class. Second, it needs a diverse training set of API mappings. Third, to find a mapped API in C#, it needs to search in a large number of candidates. Finally, it might not work for the pairs of libraries with much different paradigms.

VIII. Related Work

DeepAPI [22] uses Recurrent Neural Network Encoder-Decoder to generate API sequences for a given text by using word embeddings and deep learning. DeepAPI has a different goal as it learns the association of API sequences and annotated words to *generate API sequences from texts*. We use Word2Vec to capture the relations between API elements and generate C# API sequences from Java API sequences. While DeepAPI uses a deep learning translation model between texts and API sequences, we use transformation between two vector spaces to derive the API mappings between two languages.

API2API is also related to the work by Ye *et al.* [23]. The authors used Skip-gram model on API and reference documents and tutorials to create embeddings for words and code elements. Semantic similarities among such documents are modeled via those embeddings. The first key difference is that they aimed to quantify the relations between words and code elements, while we focus on the relations among API elements in API usages. Thus, they used documentation, while we work on API sequences from source code. Finally, their application is to improve text-code retrieval, while we support the applications involving API usages and code migration.

Researchers have applied *statistical NLP methods* including word embeddings to software artifacts. PAM [24] is a parameter-free, probabilistic algorithm to mine API patterns. It uses a probabilistic formulation of frequent sequence mining on API sequences. Allamanis *et al.* [25] suggest methods/classes' names using embeddings. The elements with statistical cooccurrences are projected into a continuous space with the words from the names. In comparison, we use Word2Vec and learn the transformation between two spaces. Their model works in the same space. Maddison and Tarlow use probabilistic CFGs and neuro-probabilistic language models for code [26].

Researchers have proposed to use language models to suggest next tokens or API calls [3], [4], [5], [27]. Allamanis *et al.* [28] use bimodal modeling for short texts and source code snippets. They use traditional probabilistic model, and we use Word2Vec for learning API embeddings. NATURALIZE [29] suggests natural identifier names and formatting conventions. API2API is inspired from a work by Mikolov *et al.* [18] where similar geometric arrangements were observed in English and Spanish words for numbers and animals. Anycode [30] uses a probabilistic CFG for Java constructs to synthesize Java expressions. SWIM [31] synthesizes code by using IBM Model to produce code elements and then uses n-gram for synthesis.

Peng *et al.* [32] propose a deep learning model to learn vector representations for tree-based source code. Mou *et al.* [33] introduce convolutional neural networks over tree structures. We could use those models in place of Word2Vec in API2API.

To mine *API mappings*, MAM [15] uses API Transformation Graphs, and compares APIs via similar names and calling structures. HiMa [20] and Aura [19] use call dependency and text similarity to identify change rules. Rosetta [16] needs pairs of functionally-equivalent applications.

Our work StaMiner [11] mines API mappings by maximizing the likelihoods of observing the mappings between API pairs from a parallel corpus of client code. The resulting API mappings are useful for rule-based migration tools [17], [34], [35], [36], [37], [38], [39]. Our prior work SLAMC [40] provides a code representation for our *phrase-based SMT models* in mppSMT [41] and semSMT [42]. Phrase-based SMT was enhanced with grammar structures [43]. SMT is used to create pseudo-code [44]. Early work on API2API was in a poster [45].

IX. Conclusion

We have shown that Word2Vec for APIs can capture the regularities of the relations of APIs in API usages. We demonstrate its usefulness in 3 applications. We build a tool to mine the pairs of API elements that share the same usage relations among them. We also propose an approach to automatically mine API mappings by learning the transformation between the two vector spaces of APIs in the source and target languages. Our experiment shows that for just one suggestion, our approach is able to achieve high precision and recall. In the final application, we build a migration tool that migrate API usages between Java and C# and show its high accuracy.

X. 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] T. T. Nguyen, H. A. Nguyen, N. H. Pham, J. M. Al-Kofahi, and T. N. Nguyen, "Graph-based mining of multiple object usage patterns," in *Proceedings of the the 7th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on The Foundations of Software Engineering*, ser. ESEC/FSE '09. New York, NY, USA: ACM, 2009, pp. 383–392. [Online]. Available: http://doi.acm.org/10.1145/1595696.1595767

[2] H. Zhong, T. Xie, L. Zhang, J. Pei, and H. Mei, "MAPO: Mining and recommending API usage patterns," in *Proceedings of the 23rd European Conference on ECOOP 2009 — Object-Oriented Programming*, ser. Genoa. Berlin, Heidelberg: Springer-Verlag, 2009, pp. 318–343. [Online]. Available: http://dx.doi.org/10.1007/978-3-642-03013-0_15

[3] 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*, ser. ICSE '12. Piscataway, NJ, USA: IEEE Press, 2012, pp. 837–847. [Online]. Available: http://dl.acm.org/citation.cfm?id=2337223.2337322

[4] 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*, ser. PLDI '14. New York, NY, USA: ACM, 2014, pp. 419–428. [Online]. Available: http://doi.acm.org/10.1145/2594291.2594321

[5] M. White, C. Vendome, M. Linares-Vásquez, and D. Poshyvanyk, "Toward deep learning software repositories," in *Proceedings of the 12th Working Conference on Mining Software Repositories*, ser. MSR '15. Piscataway, NJ, USA: IEEE Press, 2015, pp. 334–345. [Online]. Available: http://dl.acm.org/citation.cfm?id=2820518.2820559

[6] A. T. Nguyen and T. N. Nguyen, "Graph-based statistical language model for code," in *Proceedings of the 37th International Conference on Software Engineering - Volume 1*, ser. ICSE '15. Piscataway, NJ, USA: IEEE Press, 2015, pp. 858–868. [Online]. Available: http://dl.acm.org/citation.cfm?id=2818754.2818858

[7] T. Mikolov, I. Sutskever, K. Chen, G. S. Corrado, and J. Dean, "Distributed representations of words and phrases and their compositionality," in *Advances in Neural Information Processing Systems 26: 27th Annual Conference on Neural Information Processing Systems 2013 (NIPS'13).*, 2013, pp. 3111–3119.

[8] I. Jolliffe, *Principal Component Analysis*. USA: Springer-Verlag, 1986.

[9] T. Mikolov, W. T. Yih, and G. Zweig, "Linguistic regularities in continuous space word representations," in *Proceedings of the 2013 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies (NAACL-HLT-2013)*. Association for Computational Linguistics, May 2013.

[10] T. Mikolov, K. Chen, G. Corrado, and J. Dean, "Efficient estimation of word representations in vector space," *CoRR*, vol. abs/1301.3781, 2013. [Online]. Available: http://arxiv.org/abs/1301.3781

[11] 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*, ser. ASE '14. New York, NY, USA: ACM, 2014, pp. 457–468. [Online]. Available: http://doi.acm.org/10.1145/2642937.2643010

[12] D. Cer, M. Galley, D. Jurafsky, and C. D. Manning, "Phrasal: A toolkit for statistical machine translation with facilities for extraction and incorporation of arbitrary model features," in *Proceedings of the NAACL HLT 2010 Demonstration Session*, ser. HLT-DEMO '10. Stroudsburg, PA, USA: Association for Computational Linguistics, 2010, pp. 9–12. [Online]. Available: http://dl.acm.org/citation.cfm?id=1855450.1855453

[13] L. Deng and D. Yu, "Deep learning: Methods and applications," *Found. Trends Signal Process.*, vol. 7, pp. 197–387, Jun. 2014. [Online]. Available: http://dx.doi.org/10.1561/2000000039

[14] M. Allamanis and C. Sutton, "Mining source code repositories at massive scale using language modeling," in *Proceedings of the 10th Working Conference on Mining Software Repositories*, ser. MSR '13. Piscataway, NJ, USA: IEEE Press, 2013, pp. 207–216. [Online]. Available: http://dl.acm.org/citation.cfm?id=2487085.2487127

[15] H. Zhong, S. Thummalapenta, T. Xie, L. Zhang, and Q. Wang, "Mining API mapping for language migration," in *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering - Volume 1*, ser. ICSE '10. New York, NY, USA: ACM, 2010, pp. 195–204. [Online]. Available: http://doi.acm.org/10.1145/1806799.1806831

[16] A. Gokhale, V. Ganapathy, and Y. Padmanaban, "Inferring likely mappings between APIs," in *Proceedings of the International Conference on Software Engineering*, ser. ICSE '13. IEEE, 2013, pp. 82–91. [Online]. Available: http://dl.acm.org/citation.cfm?id=2486788.2486800

[17] "Java2CSharp," http://sourceforge.net/projects/j2cstranslator/.

[18] T. Mikolov, Q. V. Le, and I. Sutskever, "Exploiting similarities among languages for machine translation." *CoRR*, vol. abs/1309.4168, 2013. [Online]. Available: http://dblp.uni-trier.de/db/journals/corr/corr1309.html#MikolovLS13

[19] W. Wu, Y.-G. Guéhéneuc, G. Antoniol, and M. Kim, "Aura: A hybrid approach to identify framework evolution," in *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering - Volume 1*, ser. ICSE '10. New York, NY, USA: ACM, 2010, pp. 325–334. [Online]. Available: http://doi.acm.org/10.1145/1806799.1806848

[20] 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*, ser. ICSE '12. Piscataway, NJ, USA: IEEE Press, 2012, pp. 353–363. [Online]. Available: http://dl.acm.org/citation.cfm?id=2337223.2337265

[21] "Jv2cs project's website," http://home.eng.iastate.edu/∼trong/projects/jv2cs/.

[22] X. Gu, H. Zhang, D. Zhang, and S. Kim, "Deep API learning," in *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, ser. FSE 2016. New York, NY, USA: ACM, 2016, pp. 631–642. [Online]. Available: http://doi.acm.org/10.1145/2950290.2950334

[23] 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*, ser. ICSE '16. ACM, 2016, pp. 404–415. [Online]. Available: http://doi.acm.org/10.1145/2884781.2884862

[24] J. M. Fowkes and C. A. Sutton, "Parameter-free probabilistic API mining at GitHub scale," *CoRR*, vol. abs/1512.05558, 2015. [Online]. Available: http://arxiv.org/abs/1512.05558

[25] M. Allamanis, E. T. Barr, C. Bird, and C. Sutton, "Suggesting accurate method and class names," in *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*, ser. ESEC/FSE 2015. New York, NY, USA: ACM, 2015, pp. 38–49. [Online]. Available: http://doi.acm.org/10.1145/2786805.2786849

[26] C. J. Maddison and D. Tarlow, "Structured generative models of natural source code," in *The 31st International Conference on Machine Learning (ICML)*, June 2014.

[27] Z. Tu, Z. Su, and P. Devanbu, "On the localness of software," in *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*, ser. FSE 2014. New York, NY, USA: ACM, 2014, pp. 269–280. [Online]. Available: http://doi.acm.org/10.1145/2635868.2635875

[28] M. Allamanis, D. Tarlow, A. D. Gordon, and Y. Wei, "Bimodal modelling of source code and natural language," in *Proceedings of the 32Nd International Conference on International Conference on Machine Learning - Volume 37*, ser. ICML'15. JMLR.org, 2015, pp. 2123–2132. [Online]. Available: http://dl.acm.org/citation.cfm?id=3045118.3045344

[29] M. Allamanis, E. T. Barr, C. Bird, and C. Sutton, "Learning natural coding conventions," in *Proceedings of the 22Nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*, ser. FSE 2014. New York, NY, USA: ACM, 2014, pp. 281–293. [Online]. Available: http://doi.acm.org/10.1145/2635868.2635883

[30] 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*, ser. OOPSLA 2015. New York, NY, USA: ACM, 2015, pp. 416–432. [Online]. Available: http://doi.acm.org/10.1145/2814270.2814295

[31] M. Raghothaman, Y. Wei, and Y. Hamadi, "SWIM: Synthesizing What I Mean: Code Search and Idiomatic Snippet Synthesis," in *Proceedings of the 38th International Conference on Software Engineering*, ser. ICSE '16. New York, NY, USA: ACM, 2016, pp. 357–367. [Online]. Available: http://doi.acm.org/10.1145/2884781.2884808

[32] H. Peng, L. Mou, G. Li, Y. Liu, L. Zhang, and Z. Jin, "Building program vector representations for deep learning," *Knowledge Science, Engineering and Management, Lecture Notes in Computer Science*, vol. 9403, pp. 547–553, 2015.

[33] L. Mou, G. Li, L. Zhang, T. Wang, and Z. Jin, "Convolutional neural networks over tree structures for programming language processing," in *Proceedings of the Thirtieth AAAI Conference on Artificial Intelligence*, ser. AAAI'16. AAAI Press, 2016, pp. 1287–1293. [Online]. Available: http://dl.acm.org/citation.cfm?id=3015812.3016002

[34] "Sharpen," https://github.com/mono/sharpen.

[35] "DMS," http://www.semdesigns.com/Products/DMS/DMSToolkit.html.

[36] "Java to C# Converter," http://www.tangiblesoftwaresolutions.com/Product_Details/ Java_to_CSharp_Converter.html.

[37] "XES," http://www.euclideanspace.com/software/language/xes/userGuide/convert/javaToCSharp/.

[38] "Octopus.Net Translator," http://www.remotesoft.com/octopus/.

[39] "Microsoft Java Language Conversion Assistant," http://support.microsoft.com/kb/819018.

[40] 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 2013 9th Joint Meeting on Foundations of Software Engineering*, ser. ESEC/FSE 2013. New York, NY, USA: ACM, 2013, pp. 532–542. [Online]. Available: http://doi.acm.org/10.1145/2491411.2491458

[41] A. T. Nguyen, T. T. Nguyen, and T. N. Nguyen, "Divide-and-conquer approach for multi-phase statistical migration for source code (t)," in *Proceedings of the 2015 30th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, ser. ASE '15. Washington, DC, USA: IEEE Computer Society, 2015, pp. 585–596. [Online]. Available: http://dx.doi.org/10.1109/ASE.2015.74

[42] ——, "Migrating code with statistical machine translation," in *Companion Proceedings of the 36th International Conference on Software Engineering*, ser. ICSE Companion 2014. New York, NY, USA: ACM, 2014, pp. 544–547. [Online]. Available: http://doi.acm.org/10.1145/2591062.2591072

[43] S. Karaivanov, V. Raychev, and M. Vechev, "Phrase-based statistical translation of programming languages," in *Proceedings of the 2014 ACM International Symposium on New Ideas, New Paradigms, and Reflections on Programming & Software*, ser. Onward! 2014. New York, NY, USA: ACM, 2014, pp. 173–184. [Online]. Available: http://doi.acm.org/10.1145/2661136.2661148

[44] Y. Oda, H. Fudaba, G. Neubig, H. Hata, S. Sakti, T. Toda, and S. Nakamura, "Learning to generate pseudo-code from source code using statistical machine translation (t)," in *Proceedings of the 2015 30th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, ser. ASE '15. Washington, DC, USA: IEEE Computer Society, 2015, pp. 574–584. [Online]. Available: http://dx.doi.org/10.1109/ASE.2015.36

[45] 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*, ser. ICSE '16. New York, NY, USA: ACM, 2016, pp. 756–758. [Online]. Available: http://doi.acm.org/10.1145/2889160.2892661