DeepAM: Migrate APIs with Multi-modal Sequence to Sequence Learning

Xiaodong Gu[1], **Hongyu Zhang**[2], **Dongmei Zhang**[3] **and Sunghun Kim**[1]
[1]The Hong Kong University of Science and Technology, Hong Kong, China
[2]The University of Newcastle, Callaghan, Australia
[3]Microsoft Research, Beijing, China
guxiaodong1987@126.com, hongyujohn@gmail.com, dongmeiz@microsoft.com, hunkim@cse.ust.hk

Abstract

Computer programs written in one language are often required to be ported to other languages to support multiple devices and environments. When programs use language specific APIs (Application Programming Interfaces), it is very challenging to migrate these APIs to the corresponding APIs written in other languages. Existing approaches mine API mappings from projects that have corresponding versions in two languages. They rely on the sparse availability of bilingual projects, thus producing a limited number of API mappings. In this paper, we propose an intelligent system called DEEPAM for automatically mining API mappings from a large-scale code corpus without bilingual projects. The key component of DEEPAM is based on the multi-modal sequence to sequence learning architecture that aims to learn joint semantic representations of bilingual API sequences from big source code data. Experimental results indicate that DEEPAM significantly increases the accuracy of API mappings as well as the number of API mappings when compared with the state-of-the-art approaches.

1 Introduction

Programming language migration is an important task in software development [Mossienko, 2003; Hassan and Holt, 2005; Tonelli *et al.*, 2010]. A software product is often required to support a variety of devices and environments. This requires developing the software product in one language and manually porting it to other languages. This procedure is tedious and time-consuming. Building automatic code migration tools is desirable to reduce the effort in code migration.

However, current language migration tools, such as Java2CSharp[1], require users to manually define the migration rules between the respective program constructs and the mappings between the corresponding Application Programming Interfaces (APIs) that are used by the software libraries of the two languages. For example, The API *BufferedReader.read* in Java should be mapped to *StreamReader.read* in C# . Such a manual procedure is tedious and error-prone. As a result, only a small number of API mappings are produced [Zhong *et al.*, 2010].

To reduce manual effort in API migration, several approaches have been proposed to automatically mine API mappings from a software repository [Nguyen *et al.*, 2014; Pandita *et al.*, 2015; Zhong *et al.*, 2010]. For example, Nguyen et al. [2014] proposed StaMiner that applies statistical machine translation (SMT) [Koehn *et al.*, 2003] to bilingual projects, namely, projects that are released in multiple programming languages. It first aligns equivalent functions written in two languages that have similar names. Then, it extracts API mappings from the paired functions using the phrase-based SMT model [Koehn *et al.*, 2003].

However, existing approaches rely on the sparse availability of bilingual projects. The number of available bilingual projects is often limited due to the high cost of manual code migration. For example, we analyzed 11K Java projects on GitHub which were created between 2008 to 2014. Among them, only 15 projects have been manually ported to C# versions. Therefore, the number of API mappings produced by existing approaches is rather limited. In addition, given bilingual projects, they need aligning equivalent functions using name similarity heuristics. Only a portion of functions in a bilingual project have similar function names and can be aligned [Zhong *et al.*, 2010].

In this paper, we propose DEEPAM (Deep API Migration), a novel, deep learning based system to API migration. Without the restriction of using bilingual projects, DEEPAM can directly identify equivalent source and target API sequences from a large-scale commented code corpus. The key idea of DEEPAM is to learn the semantic representations of both source and target API sequences and identify semantically related API sequences for the migration. DEEPAM assigns to each API sequence a continuous vector in a high-dimensional semantic space in such a way that API sequences with similar vectors, or "embeddings", tend to have similar natural language descriptions.

In our approach, DEEPAM first extracts API sequences (i.e., sequences of API invocations) from each function in the code corpus. For each API sequence, it assigns a natural language description that is automatically extracted from corresponding code comments. With the ⟨API sequence, description⟩ pairs, DEEPAM applies the sequence-to-sequence learning [Cho *et al.*, 2014] to embed each API

[1]http://j2cstranslator.wiki.sourceforge.net/

sequence into a fixed-length vector that reflects the intent in the corresponding natural language description. By *jointly embedding* both source and target API sequences into the same space, DEEPAM aligns the equivalent source and target API sequences that have the closest embeddings. Finally, the pairs of aligned API sequences are used to extract general API mappings using SMT.

To our knowledge, DEEPAM is the first system that applies deep learning techniques to learn the semantic representations of API sequences from a large-scale code corpus. It has the following key characteristics that make it unique:

- **Big source code:** DEEPAM enables the construction of large-scale bilingual API sequences from big code corpus rather than limited bilingual projects. It learns API semantic representations from 10 million commented code snippets collected over seven years.
- **Deep model:** The multi-modal sequence-to-sequence learning architecture ensures the system can learn deeper semantic features of API sequences than the traditional shallow ones.

2 Related Work

API migration has been investigated by many researchers [Nguyen *et al.*, 2014; Pandita *et al.*, 2015; Zhong *et al.*, 2010]. Zhong et al. [2010] proposed MAM, a graph based approach to mine API mappings. MAM builds on projects that are released with multiple programming languages. It uses name similarity to align client code of both languages. Then, it detects API mappings between these functions by analyzing their API Transformation Graphs. Nguyen et al. [2014] proposed StaMiner that directly applies statistical machine translation to bilingual projects.

However, these techniques require the same client code to be available on both the source and the target platforms. Therefore, they rely on the availability of software packages that have been ported manually from the source to the target platform. Furthermore, they use name similarity as a heuristic in their API mapping algorithms. Therefore, they cannot align equivalent API sequences from client code which are similar but independently-developed.

Pandita et al. [2015] proposed TMAP, which applies the vector space model [Manning *et al.*, 2008], an information retrieval technique, to discover likely mappings between APIs. For each source API, it searches target APIs that have similar text descriptions in their API documentation. However, the vector space model they applied is based on the bag-of-words assumption; it cannot identify sentences with semantically related words and with different sequences of words.

Recently, deep learning technology [Sutskever *et al.*, 2014; Cho *et al.*, 2014] has been shown to be highly effective in various domains (e.g., computer vision and natural language processing). Researchers have begun to apply this technology to tackle some software engineering problems. Huo et al. propose a neural model to learn unified features from natural and programming languages for locating buggy source code [Huo *et al.*, 2016]. Gu et al. apply sequence-to-sequence learning to generate API sequences from natural language queries [Gu *et al.*, 2016]. Hence, this study constitutes the first attempt to apply the deep learning approach to migrate APIs between two programming languages.

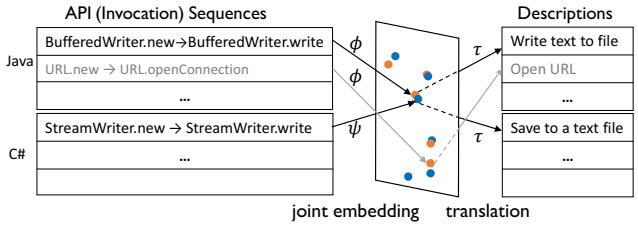

Figure 1: An Illustration of Joint Semantic Embedding

3 Method

Let $\mathcal{A}$=$\{a^{(i)}\}$ denote a set of API sequences where $a^{(i)}$=$[\alpha_1, ..., \alpha_{L_a}]$ denotes the sequence of API invocations in a function. Suppose we are given a set of source API sequences $\mathcal{A}_S$=$\{a_S^{(i)}\}$ (i.e., API sequences in a source language) and a set of target API sequences $\mathcal{A}_T$=$\{a_T^{(i)}\}$ (i.e., API sequences in a target language). Our goal is to find an alignment between $\mathcal{A}_S$ and $\mathcal{A}_T$, namely,

$$f : \mathcal{A}_S \rightarrow \mathcal{A}_T \tag{1}$$

so that each source API sequence $a_S^{(i)} \in \mathcal{A}_S$ is mapped to an equivalent target API sequence $a_T^{(j)} \in \mathcal{A}_T$.

Since $\mathcal{A}_S$ and $\mathcal{A}_T$ are heterogeneous, it is difficult to discover the correlation f directly. Our approach is based on the intuition of **"third party translation"**. That is, although $\mathcal{A}_S$ and $\mathcal{A}_T$ are heterogeneous, in the sense of vocabulary and usage patterns, they can both be mapped to high-level user intents described in natural language. Thus, we can bridge them through their natural language descriptions. For each $a^{(i)} \in \mathcal{A}$, we assume that there is a corresponding natural language description $d^{(i)}$=$[w_1, ..., w_{L_d}]$ represented as a sequence of words.

The idea can be formulated with Joint Embedding(a.k.a., multi-modal embedding) [Xu *et al.*, 2015], a technique to jointly embed/correlate heterogeneous data into a unified vector space so that semantically similar concepts across the two modalities occupy nearby regions of the space [Andrej and Li, 2015]. In our approach, the joint embedding of $\mathcal{A}_S$ and $\mathcal{A}_T$ can be formulated as:

$$\mathcal{A}_S \xrightarrow{\phi} V \xrightarrow{\tau} \mathcal{D} \xleftarrow{\tau} V \xleftarrow{\psi} \mathcal{A}_T \tag{2}$$

where $V \in \mathbb{R}^d$ is a common vector space representing the semantics of API sequences; ϕ:$\mathcal{A}_S \rightarrow V$ is an embedding function to map $\mathcal{A}_S$ into V, ψ:$\mathcal{A}_T \rightarrow V$ is an embedding function to map $\mathcal{A}_T$ into V, $\mathcal{D}$=$\{d_S^{(i)}\} \cup \{d_T^{(i)}\}$ is the space of natural language descriptions. $\tau : V \rightarrow \mathcal{D}$ is a function to translate from the semantic representations V to corresponding natural language descriptions $\mathcal{D}$.

Through joint embedding, $\mathcal{A}_S$ and $\mathcal{A}_T$ can be easily correlated through their semantic vectors $V_{\mathcal{A}_S}$ and $V_{\mathcal{A}_T}$. Figure 1 shows an illustration of joint semantic embedding between Java and C# API sequences. We are given a corpus of API sequences (in both Java and C#) and the corresponding natural language descriptions. Each API sequence is embedded (through ϕ or ψ) and translated (through τ) to its corresponding description. The yellow and blue

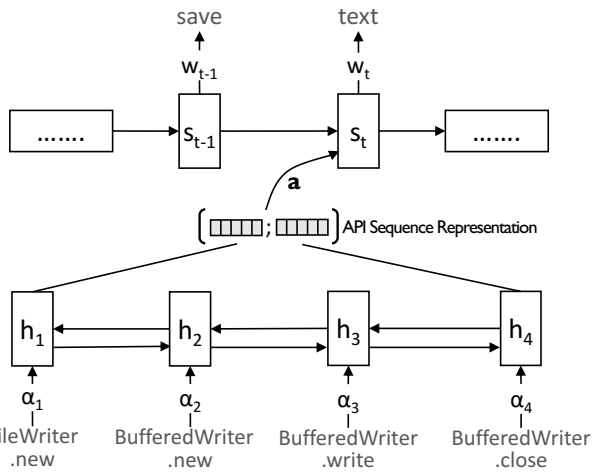

Figure 2: The sequence-to-sequence learning framework for API Semantic Embedding. A bidirectional RNN is used to concatenate the forward and backward hidden states as the semantic representations of API sequences

points represent embeddings of Java and C# APIs respectively. Through traning, the Java API sequence *BufferedWriter.new→ BufferedWriter.write* and the C# API sequence *StreamWriter.new→StreamWriter.write* are embedded into a nearby place in order to generate similar corresponding descriptions *write text to file* and *save to a text file*. Therefore, the two API sequences can be identified as semantically equivalent API sequences.

3.1 Learning Semantic Representations of API Sequences

In our approach, the semantic embedding function (ϕ or ψ) and the translation function τ are realized using the RNN-based sequence-to-sequence learning framework [Cho *et al.*, 2014]. The sequence-to-sequence learning is a general framework where the input sequence is embedded to a vector that represents the semantic representation of the input, and the semantic vector is then used to generate the target sequence. The model that embeds the sequence to a vector (i.e., ϕ or ψ) is called "encoder", and the model that generates the target sequence(i.e., τ) is called "decoder".

The framework of the sequence-to-sequence model applied to API semantic embedding is illustrated in Figure 2. Given a set of ⟨API sequence, description⟩ pairs $\{\langle a^{(i)}, d^{(i)}\rangle\}$, The encoder (a bi-directional recurrent neural network [Mikolov *et al.*, 2010]) converts each API sequence a=$[\alpha_1, ..., \alpha_{L_a}]$, to a fixed-length vector $\boldsymbol{a}$ using the following equations iteratively from t = 1 to L_d:

$$\boldsymbol{h}_t = tanh(\boldsymbol{W}_{enc}[\boldsymbol{h}_{t-1}; \boldsymbol{\alpha}_t] + \boldsymbol{b}_{enc}) \quad (3)$$

$$\boldsymbol{a} = \boldsymbol{h}_{L_a} \quad (4)$$

where $\boldsymbol{h}_t(t$=$1, ..., L_a)$ represents the hidden states of the RNN at each potion t of the input; $[a;b]$ represents the concatenation of two vectors, $\boldsymbol{W}_{enc}$ and $\boldsymbol{b}_{enc}$ are trainable parameters in the RNN, $tanh$ is the activation function.

The decoder then uses the encoded vector to generate the corresponding natural language description d by sequentially predicting a word w_t conditioned on the vector $\boldsymbol{a}$ as well as previous words $w_1, ..., w_{t-1}$.

$$Pr(d) = \prod_{t=1}^{L_d} p(w_t | w_1, ..., w_{t-1}, \boldsymbol{a}) \quad (5)$$

$$p(w_t | w_1, ..., w_{t-1}, \boldsymbol{a}) = \mathrm{softmax}(\boldsymbol{W}^o_{dec}\boldsymbol{s}_t + \boldsymbol{b}^o_{dec}) \quad (6)$$

$$\boldsymbol{s}_t = tanh(\boldsymbol{W}^s_{dec}[\boldsymbol{s}_{t-1}; \boldsymbol{w}_{t-1}; \boldsymbol{a}] + \boldsymbol{b}^s_{dec}) \quad (7)$$

where $\boldsymbol{s}_t(t$=$1, ..., L_d)$ represents the hidden states of the RNN at each potion t of the output; $\boldsymbol{W}^o_{dec}$, $\boldsymbol{b}^o_{dec}$, $\boldsymbol{W}^s_{dec}$ and $\boldsymbol{b}^s_{dec}$ are trainable parameters in the decoder RNN.

Both the encoder and decoder RNNs are implemented as a bidirectional gated recurrent neural network (GRU) [Cho *et al.*, 2014] which is a widely used implementation of RNN. Both GRUs have two hidden layers, each with 1000 hidden units.

3.2 Joint Semantic Embedding for Aligning Equivalent API Sequences

For *joint embedding*, we train the sequence-to-sequence model on both $\{\langle a^{(i)}_S, d^{(i)}_S\rangle\}$ and $\{\langle a^{(i)}_T, d^{(i)}_T\rangle\}$ to minimize the following objective function:

$$\begin{aligned}\mathcal{L}(\theta) = &-\frac{1}{N_S}\sum_{i=1}^{N_S}\sum_{t=1}^{L_d}\log p_\theta(w^{(it)}_S | a^{(i)}_S) \\ &-\frac{1}{N_T}\sum_{i=1}^{N_T}\sum_{t=1}^{L_d}\log p_\theta(w^{(it)}_T | a^{(i)}_T)\end{aligned} \quad (8)$$

where N_S and N_T are the total number of source and target training instances, respectively. L_d is the length of each natural language sentence. θ denotes model parameters, while $p_\theta(w^{(it)}|a^{(i)})$ (derived from Equation 3 to 7) denotes the likelihood of generating the t-th target word given the API sequence $a^{(i)}$ according to the model parameters θ.

After training, each API sequence a=$[\alpha_1, ..., \alpha_{L_a}]$ is embedded to a vector $\boldsymbol{a}$ that reflects developer's high-level intent. We identify equivalent source and target API sequences as those having close semantic vectors.

4 Implementation

In this section, we describe the detailed implementation of DEEPAM, a deep-learning based system we propose to migrate API usage sequences. Figure 3 shows the overall workflow of DEEPAM. It includes four main steps. We first prepare a large-scale corpus of ⟨API sequence, description⟩ pairs for both Java and C# (Step 1). The pairs of both languages are jointly embedded by the sequence-to-sequence model as described in Section 3.2 (Step 2). Then, we identify related Java and C# API sequences according to their semantic vectors (Step 3). Finally, a statistical machine translation component is used to extract general API mappings from the aligned bilingual API sequences (Step 4).

In theory, our system could migrate APIs between any programming languages. In this paper we limit our scope to the Java-to-C# migration. The details of each step are explained in the following sections.

4.1 Gathering a Large-scale API Sequence-to-Description Corpus

We first construct a large-scale database that contains ⟨API sequence, description⟩ pairs for training the model. We down-

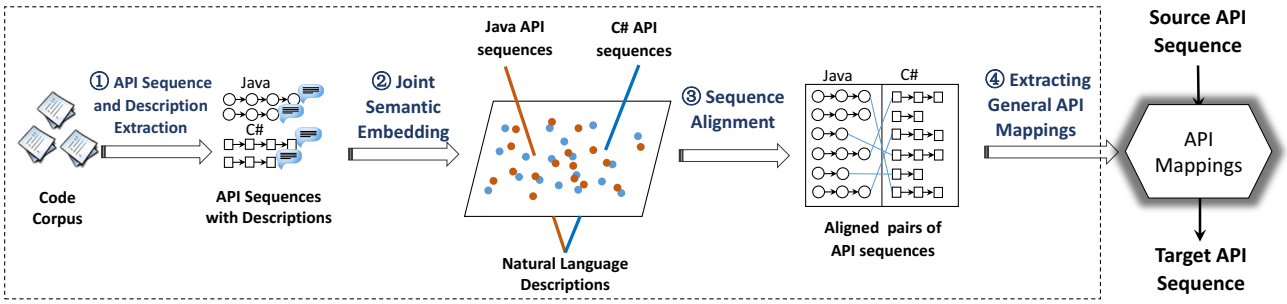

Figure 3: The Overall Workflow of DEEPAM

```
/// <summary>
/// Get the content of the file.
/// </summary>
/// <param name="sFilename">File path and name.</param>
///
public static string ReadFile(String sFilename) {
   StreamReader myFile
         = new StreamReader(sFilename, System.Text.Encoding.Default);
   string sContent = myFile.ReadToEnd();
   myFile.Close();
   return sContent;
}
```

API sequence: StreamReader.new→StreamReader.ReadToEnd→StreamReader.Close
Description: get the content of the file.

Figure 4: An example of extracting an API sequence and its description from a C# function *TextFile.ReadFile*[7]

load Java and C# projects created from 2008 to 2014 from GitHub[2]. To remove toy or experimental programs, we only select the projects with at least one star. In total, we collected 442,928 Java projects and 182,313 C# projects from GitHub.

Having collected the code corpus, we extract API sequences and corresponding natural language descriptions: we parse source code files into ASTs (Abstract Syntax Trees) using Eclipse's JDT compiler[3] for Java projects, and Roslyn[4] for C# projects. Then, we extract the API sequence from individual functions using the same approach in [Gu *et al.*, 2016].

To obtain natural language descriptions for the extracted API sequences, we extract function-level code summaries from code comments. In both Java and C#, it is the first sentence of a *documentation comment*[5] for a function. According to the Javadoc guidance[6], the first sentence of a documentation comment is used as a short summary of a function. Figure 4 shows an example of *documentation comments* for a C# function *TextFile.ReadFile*[7] in the Gitlab CI project.

Finally, we obtain a database consisting of 9,880,169 ⟨API sequence, description⟩ pairs, including 5,271,526 Java pairs and 4,608,643 C# pairs.

[2]http://github.com

[3]http://www.eclipse.org/jdt

[4]https://roslyn.codeplex.com/

[5]A *documentation comment* in Java starts with '/**' and ends with '*/'. A *documentation comment* in C# starts with a "<summary>" tag and ends with a "</summary>" tag.

[6]http://www.oracle.com/technetwork/articles/java/index-137868.html

[7]https://github.com/virtualmarc/gitlab-ci-runner-win/blob/master/gitlab-ci-runner/helper/TextFile.cs

4.2 Model Training

We train the sequence-to-sequence model on the collected ⟨API sequence, description⟩ pairs of both Java and C#. The model is trained using the mini-batch stochastic gradient descent algorithm (SGD) [Bottou, 2010] together with Adadelta [Zeiler, 2012]. We set the batch size as 200. Each batch is constituted with 100 Java pairs and 100 C# pairs that are randomly selected from corresponding datasets. The vocabulary sizes of both APIs and natural language descriptions are set to 10,000. The maximum sequence lengths L_a and L_d are both set as 30. Sequences that exceed the maximum lengths will be excluded for training.

After training, we feed in the encoder with all API sequences and obtain corresponding semantic vectors from the last hidden layer of encoder.

4.3 API Sequence Alignment

After embedding all API sequences, we build pairs of equivalent Java and C# API sequences according to their semantic vectors. For each Java API sequence, we find the most related C# API sequence to align with by selecting the C# API sequence that has the most similar vector representation. We measure the similarity between the vectors of two API sequences using the cosine similarity, which is defined as:

$$\text{similarity}(\boldsymbol{a}_s, \boldsymbol{a}_t) = \frac{\boldsymbol{a}_s \cdot \boldsymbol{a}_t}{\| \boldsymbol{a}_s \| \| \boldsymbol{a}_t \|} \quad (9)$$

where $\boldsymbol{a}_s$ and $\boldsymbol{a}_t$ are vectors of source and target API sequences. The higher the similarity, the more related the source and target API sequences are to each other.

Finally, we obtain a database consisting of aligned pairs of Java and C# API sequences.

4.4 Extracting General API Mappings

The aligned pairs of API sequences may be project-specific. However, automated code migration tools such as Java2CSharp require commonly used API mappings. To obtain more general API mappings, we summarize mappings that have high co-occurrence probabilities in the aligned pairs of API sequences. To do so, we apply an SMT technique named *phrase-based model* [Koehn *et al.*, 2003] to the pairs of aligned API sequences. The phrase-based model was originally designed to extract phrase-to-phrase translation mappings from bilingual sentences. In our system, the phrase model summarizes pairs of API phrases, namely, subsequences of APIs that frequently co-occur in the aligned pairs of API sequences. For each phrase pair, it assigns a score defined as the translation probability

Table 1: Accuracy of 1-to-1 API mappings mined by DEEPAM and StaMiner (%)

Package	Class Migration						Method Migration					
	Precision		Recall		F-score		Precision		Recall		F-score	
	StaMiner	DeepAM	StaMiner	DeepAM	StaMiner	DeepAM	StaMiner	DeepAM	StaMiner	DeepAM	StaMiner	DeepAM
java.io	70.0%	80.0%	63.6%	75.0%	66.6%	72.7%	70.0%	66.7%	64.0%	87.5%	66.9%	75.2%
java.lang	82.5%	80.0%	76.7%	81.3%	79.5%	80.7%	86.7%	83.7%	76.5%	87.2%	81.3%	85.4%
java.math	50.0%	66.7%	50.0%	66.7%	50.0%	66.7%	66.7%	66.7%	66.7%	66.7%	66.7%	66.7%
java.net	100.0%	100.0%	50.0%	100.0%	66.7%	100.0%	100.0%	100.0%	33.3%	100.0%	50.0%	100.0%
java.sql	100.0%	100.0%	50.0%	100.0%	66.7%	100.0%	100.0%	50.0%	50.0%	66.7%	66.7%	57.2%
java.util	64.7%	69.6%	71.0%	72.7%	67.7%	71.1%	63.0%	64.3%	54.8%	85.7%	58.6%	73.5%
All	77.9%	**82.7%**	60.2%	**82.6%**	66.2%	**81.9%**	81.1%	71.9%	57.6%	**82.3%**	65.0%	**76.3%**

$p(t|s)$=$\mathrm{count}(s,t)/(\mathrm{count}(s)$+1), where $\mathrm{count}(s,t)$ is the number of mapping occurrences $s \rightarrow t$, and $\mathrm{count}(s)$ is the number of all occurrences of the subsequence s. Finally, we select pairs whose translation probabilities are greater than a threshold as the final API mappings. We set the threshold to 0.5 as in StaMiner [Nguyen *et al.*, 2014]

5 Experimental Results

5.1 Accuracy in Mining API Mappings

We first evaluate how accurate DEEPAM performs in mining API mappings. We focus on 1-to-1 API mappings that are currently used by many code migration tools such as Java2Csharp. We compare the 1-to-1 API mappings mined by DEEPAM (Section 4) with a ground truth set of manually written API mappings provided by Java2CSharp.

Metric We use the F-score to measure the accuracy. It is defined as: F=$2PR/(P$+$R)$ where P=$\frac{TP}{TP+FP}$ and R=$\frac{TP}{TP+FN}$. *TP* is true positive, namely, the number of API mappings that are both in DEEPAM results and in the ground truth set. *FP* is false positive which represents the number of resulting mappings that are not in the ground truth set. *FN* is false negative, which represents the number of mappings that are in the ground truth set but not in the results.

Baselines We compare DEEPAM with StaMiner [Nguyen *et al.*, 2014] and TMAP [Pandita *et al.*, 2015]. StaMiner is a state-of-the-art API migration approach that directly utilizes statistical machine translation on bilingual projects. TMAP [Pandita *et al.*, 2015] is an API migration approach using information retrieval techniques. It aligns Java and C# APIs by searching similar descriptions in API documentation. For easy comparison, we use the same configuration as in TMAP [Pandita *et al.*, 2015]. We manually examine the numbers of correctly mined API mappings on several Java SDK classes and make a direct comparison with the TMAP's results presented in their paper.

Results Table 1 shows the accuracy of both DEEPAM and StaMiner. We evaluate the accuracy of mappings for both API classes and API methods. The results show that DEEPAM is able to mine more correct API mappings. It achieves average recalls of 82.6% and 82.3% for class and method migrations respectively, which are significantly greater than StaMiner (60.2% and 57.6%). The average precisions of DEEPAM are 82.7% and 71.9%, slightly less than but similar to StaMiner (77.9% and 81.1%). Overall, DEEPAM performs better than StaMiner, with average F-measures of 81.9% and 76.3% compared to StaMiner 's (66.2% and 65.0%).

Table 2 shows the number of correctly mined API mappings by TMAP and DEEPAPI. The column # Methods lists the total numbers of API methods for each class. As shown in the results, DEEPAM can mine many more correct API mappings than TMAP, which is based on text similarity matching.

Table 2: Number of correct API mappings mined by DEEPAM and TMAP

Class	# Methods	# API mappings	
		TMAP	DEEPAM
java.io.File	54	26	43
java.io.Reader	10	6	8
java.io.Writer	10	10	7
java.util.Calendar	47	5	20
java.util.Iterator	3	1	3
java.util.HashMap	17	5	14
java.util.ArrayList	28	15	26
java.sql.Connection	52	13	23
java.sql.ResultSet	187	31	33
java.sql.Statement	42	5	15
All	450	117	192

Table 3: Number of API Mappings Mined by DEEPAM and StaMiner

Tool	# API Mappings by Sequence Length					Corr.	EDR
	1	2-3	4-7	8+	Total		
StaMiner	50,992	31,754	14,370	3,708	100,825	87.1%	7.3%
DEEPAM	35,973	218,957	328,290	225,268	808,488	88.2%	8.2%

The results indicate that without the restriction of a few bilingual projects, DEEPAM yields many more correct API mappings.

5.2 The Scale of Mined API Mappings

We also evaluate the scalability of DEEPAM on mining API mappings: we compare the number of API mappings minend by DEEPAM and StaMiner [Nguyen *et al.*, 2014] with respect to sequence lengths. We can make this comparison because both DeepAM and StaMiner support sequence-to-sequence mapping. We also consider the quality of mined API mappings in the comparison. We use *correctness* and *edit distance ratio* (EDR) to measure the quality as used in [Nguyen *et al.*, 2014]. The *correctness* is defined as the percentage of correct API sequences of all the migrated results. The *EDR* is defined as the ratio of elements that a user must delete/add in order to transform a result into a correct one. EDR=$\frac{\sum_{\text{pairs}} \mathrm{EditDist}(s_R, s_T)}{\sum_{\text{pairs}} \mathrm{length}(s_T)}$, where $\mathrm{EditDist}$ measures the edit distance between the ground truth sequence s_R and the result sequence s_T; $\mathrm{length}(s_T)$ is the number of symbols in s_T. The value of EDR ranges from 0 to 100%. The smaller the better.

Results Table 3 shows the number of API mappings produced by DEEPAM and StaMiner. Each column within # API Mappings by Sequence Length shows the number of mined API mappings within a specific range of length: one (column 1), two or three (2-3), four to seven (4-7), and eight or more (8+).

Table 4: Examples of Mined API Mappings

Task	Java API Sequence	Migrated C# API sequence by DEEPAM
parse datetime from string	SimpleDateFormat.new SimpleDateFormat.parse	DateTimeFormatInfo.new DateTime.parseExact DateTime.parse
open a url	URL.new URL.openConnection	WebRequest.create Uri.new HttpWebRequest.getRequestStream
get files in folder	File.new File.list File.new File.isDirectory	DirectoryInfo.new DirectoryInfo.getDirectories
generate md5 hash code	MessageDigest.getInstance MessageDigest.update MessageDigest.digest	MD5.create UTF8Encoding.new UTF8Encoding.getBytes MD5.computeHash
execute sql statement	Connection.prepareStatement PreparedStatement.execute	SqlConnection.open SqlCommand.new SqlCommand.executeReader
create directory	File.new File.exists File.createNewFile	FileInfo.new Directory.exists Directory.createDirectory
read file	System.getProperty FileInputStream.new InputStreamReader.new BufferedReader.new BufferedReader.read BufferedReader.close	FileInfo.new StreamReader.new StreamReader.read StreamReader.close
create socket	InetSocketAddress.new ServerSocket.new ServerSocket.bind ServerSocket.close	Socket.new IPEndPoint.new Socket.bind Socket.close
download file from url	URL.new URL.openConnection URLConnection.getInputStream BufferedInputStream.new	WebRequest.create HttpWebRequest.getResponse HttpWebResponse.getResponseStream StreamReader.new
save an image to a file	BufferedImage.new Color.new Color.getRGB BufferedImage.setRGB String.endsWith File.new ImageIO.write	Bitmap.new Color.new Color.fromArgb Bitmap.setPixel Bitmap.save
parse xml	DocumentBuilderFactory.newInstance DocumentBuilderFactory.newDocumentBuilder DocumentBuilder.parse	XDocument.load HttpUtility.htmlEncode XDocument.parse
play audio	AudioSystem.getClip File.new AudioSystem.getAudioInputStream Clip.open Clip.start Clip.isRunning Thread.sleep Clip.close	SoundPlayer.new SoundPlayer.play Thread.sleep SoundPlayer.stop

Table 5: Accuracy of API pair alignment by DEEPAM and IR-based technique

Tool	Java version	C# version	Average
IR	37.4%	44.1%	40.8%
DEEPAM	60.2%	84.6%	72.4%

As we can see, DEEPAM produces many more API mappings than StaMiner, with comparable quality. The total number of mappings mined by DEEPAM is 808,488, which is significantly greater than that of StaMiner (100,825). In particular, DeepAM produces more mappings for long API sequences. The quality of mappings by DEEPAM is comparable to that by StaMiner. The correctness of DEEPAM is 88.2%, which is slightly greater than that of StaMiner (87.1%). However, mappings produced by DEEPAM need slightly more error correlations than StaMiner.

Overall, the results indicate that DEEPAM significantly increases the number of API mappings than StaMiner, with comparable quality. These results are expected because DEEPAM does not rely upon bilingual projects, therefore significantly increasing the size of available training corpus.

Table 4 shows some concrete examples of API mappings. We selected 12 programming tasks that are commonly used in the literature [Lv *et al.*, 2015; Gu *et al.*, 2016]. The results show that DEEPAM can successfully migrate API sequences for these tasks. DEEPAM also performs well in longer API sequences such as *read file* and *play audio*.

5.3 Effectiveness of Multi-modal API Sequence Embedding

As the most distinctive feature of our approach is the multi-modal semantic embedding of API sequences, we also evaluate DEEPAM's effectiveness in embedding API sequences, namely, whether the joint embedding is effective on API sequence alignment. As described in Section 4.3, we apply the semantic embedding and sequence alignment on raw API sequences, and obtain a database of semantically related Java and C# API sequences. We randomly select 500 aligned pairs of Java and C# API sequences from the database and manually examine whether each pair is indeed related. We calculate the ratio of related pairs of the 500 sampled pairs.

Baseline We compare our results with an IR based approach. This approach aligns API sequences by directly matching corresponding descriptions using text similarities (e.g., the vector space model) [Manning *et al.*, 2008]. We implement it using Lucene[8]. For each Java API sequence, we search the C# API sequence whose description is most similar to the description of the Java API sequence, and vice versa. We randomly select 500 aligned pairs from the results and manually examine the ratio of correctly aligned pairs.

Results Table 5 shows the performance of sequence alignment. The column Java version shows the ratio of Java API sequences which are correctly aligned to C# API sequences. Likewise, the C# version column shows the ratio of C# API sequences that are correctly aligned to Java API sequences. The results show that the joint embedding is effective for the API sequence alignment. The ratio of successful alignments is 72.4%, which significantly outperforms the IR based approach (average accuracy is 40.8%). The results indicate that the deep learning model is more effective in learning semantics of API sequences than traditional shallow models such as the vector space model.

6 Conclusion

In this paper, we propose a deep learning based approach to the migration of APIs. Without the restriction of using bilingual projects, our approach can align equivalent API sequences from a large-scale commented code corpus through multi-modal sequence-to-sequence learning. Our experimental results have shown that the proposed approach significantly increases the accuracy and scale of API mappings the state-of-the-art approaches can achieve. Our work demonstrates the effectiveness of deep learning in API migration and is one step towards automatic code migration.

References

[Andrej and Li, 2015] Karpathy Andrej and Fei-Fei Li. Deep visual-semantic alignments for generating image descriptions. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pages 3128–3137, 2015.

[8] https://lucene.apache.org/

[Bottou, 2010] Léon Bottou. Large-scale machine learning with stochastic gradient descent. In *Proceedings of COMPSTAT'2010*, pages 177–186. Springer, 2010.

[Cho *et al.*, 2014] Kyunghyun Cho, Bart Van Merriënboer, Çalar Gülçehre, Dzmitry Bahdanau, Fethi Bougares, Holger Schwenk, and Yoshua Bengio. Learning phrase representations using rnn encoder–decoder for statistical machine translation. In *Proceedings of the 2014 Conference on Empirical Methods in Natural Language Processing (EMNLP)*, pages 1724–1734, Doha, Qatar, October 2014. Association for Computational Linguistics.

[Gu *et al.*, 2016] Xiaodong Gu, Hongyu Zhang, Dongmei Zhang, and Sunghun Kim. Deep API learning. In *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, pages 631–642. ACM, 2016.

[Hassan and Holt, 2005] Ahmed E Hassan and Richard C Holt. A lightweight approach for migrating web frameworks. *Information and Software Technology*, 47(8):521–532, 2005.

[Huo *et al.*, 2016] Xuan Huo, Ming Li, and Zhi-Hua Zhou. Learning unified features from natural and programming languages for locating buggy source code. In *Proceedings of the 25th 25th International Joint Conference on Artificial Intelligence (IJCAI'16)*, 2016.

[Koehn *et al.*, 2003] Philipp Koehn, Franz Josef Och, and Daniel Marcu. Statistical phrase-based translation. In *Proceedings of the 2003 Conference of the North American Chapter of the Association for Computational Linguistics on Human Language Technology-Volume 1*, pages 48–54. Association for Computational Linguistics, 2003.

[Lv *et al.*, 2015] Fei Lv, Hongyu Zhang, Jianguang Lou, Shaowei Wang, Dongmei Zhang, and Jianjun Zhao. CodeHow: Effective code search based on API understanding and extended boolean model. In *Proceedings of the 30th IEEE/ACM International Conference on Automated Software Engineering (ASE 2015)*. IEEE, 2015.

[Manning *et al.*, 2008] Christopher D Manning, Prabhakar Raghavan, Hinrich Schütze, et al. *Introduction to information retrieval*, volume 1. Cambridge university press Cambridge, 2008.

[Mikolov *et al.*, 2010] Tomas Mikolov, Martin Karafiát, Lukas Burget, Jan Cernockỳ, and Sanjeev Khudanpur. Recurrent neural network based language model. In *INTERSPEECH 2010, 11th Annual Conference of the International Speech Communication Association, Makuhari, Chiba, Japan, September 26-30, 2010*, pages 1045–1048, 2010.

[Mossienko, 2003] Maxim Mossienko. Automated cobol to java recycling. In *Software Maintenance and Reengineering, 2003. Proceedings. Seventh European Conference on*, pages 40–50. IEEE, 2003.

[Nguyen *et al.*, 2014] Anh Tuan Nguyen, Hoan Anh Nguyen, Tung Thanh Nguyen, and Tien 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, New York, NY, USA, 2014. ACM.

[Pandita *et al.*, 2015] R. Pandita, R. P. Jetley, S. D. Sudarsan, and L. Williams. Discovering likely mappings between APIs using text mining. In *Source Code Analysis and Manipulation (SCAM), 2015 IEEE 15th International Working Conference on*, pages 231–240, Sept 2015.

[Sutskever *et al.*, 2014] Ilya Sutskever, Oriol Vinyals, and Quoc VV Le. Sequence to sequence learning with neural networks. In *Advances in neural information processing systems*, pages 3104–3112, 2014.

[Tonelli *et al.*, 2010] Thiago Tonelli, Krzysztof, and Ralf. Swing to swt and back: Patterns for API migration by wrapping. In *Software Maintenance (ICSM), 2010 IEEE International Conference on*, pages 1–10, Sept 2010.

[Xu *et al.*, 2015] Ran Xu, Caiming Xiong, Wei Chen, and Jason J Corso. Jointly modeling deep video and compositional text to bridge vision and language in a unified framework. In *AAAI*, pages 2346–2352. Citeseer, 2015.

[Zeiler, 2012] Matthew D Zeiler. Adadelta: an adaptive learning rate method. *arXiv preprint arXiv:1212.5701*, 2012.

[Zhong *et al.*, 2010] Hao Zhong, Suresh Thummalapenta, Tao Xie, Lu Zhang, and Qing Wang. Mining API mapping for language migration. In *Proceedings of the 32Nd ACM/IEEE International Conference on Software Engineering - Volume 1*, ICSE '10, pages 195–204, New York, NY, USA, 2010. ACM.