Deep API Learning

Xiaodong Gu[§], Hongyu Zhang[†], Dongmei Zhang[†], and Sunghun Kim[§]
[§]The Hong Kong University of Science and Technology, Hong Kong, China
{xguaa,hunkim}@cse.ust.hk
[†]Microsoft Research, Beijing, China
{honzhang,dongmeiz}@microsoft.com

ABSTRACT

Developers often wonder how to implement a certain functionality (e.g., how to parse XML files) using APIs. Obtaining an API usage sequence based on an API-related natural language query is very helpful in this regard. Given a query, existing approaches utilize information retrieval models to search for matching API sequences. These approaches treat queries and APIs as bags-of-words and lack a deep understanding of the semantics of the query.

We propose DeepAPI, a deep learning based approach to generate API usage sequences for a given natural language query. Instead of a bag-of-words assumption, it learns the sequence of words in a query and the sequence of associated APIs. DeepAPI adapts a neural language model named RNN Encoder-Decoder. It encodes a word sequence (user query) into a fixed-length context vector, and generates an API sequence based on the context vector. We also augment the RNN Encoder-Decoder by considering the importance of individual APIs. We empirically evaluate our approach with more than 7 million annotated code snippets collected from GitHub. The results show that our approach generates largely accurate API sequences and outperforms the related approaches.

CCS Concepts

•**Software and its engineering** → **Reusability;**

Keywords

API, deep learning, RNN, API usage, code search

1. INTRODUCTION

To implement a certain functionality, for example, how to parse XML files, developers often reuse existing class libraries or frameworks by invoking the corresponding APIs. Obtaining which APIs to use, and their usage sequence (the method invocation sequence among the APIs) is very helpful in this regard [15, 46, 49]. For example, to "*parse XML files*" using JDK library, the desired API usage sequence is as follows:

DocumentBuilderFactory.newInstance
DocumentBuilderFactory.newDocumentBuilder
DocumentBuilder.parse

Yet learning the APIs of an unfamiliar library or software framework can be a significant obstacle for developers [15, 46]. A large-scale software library such as .NET framework and JDK could contain hundreds or even thousands of APIs. In practice, usage patterns of API methods are often not well documented [46]. In a survey conducted by Microsoft in 2009, 67.6% respondents mentioned that there are obstacles caused by inadequate or absent resources for learning APIs [38]. Another field study found that a major challenge for API users is to discover the subset of the APIs that can help complete a task [39].

A common place to discover APIs and their usage sequence is from a search engine. Many developers search APIs from general web search engines such as Google and Bing. Developers can also perform a code search over an open source repository such as GitHub [3] and then utilize an API usage pattern miner [15, 46, 49] to obtain the appropriate API sequences.

However, search engines are often inefficient and inaccurate for programming tasks [42]. General web search engines are not designed to specifically support programming tasks. Developers need to manually examine many web pages to learn about the APIs and their usage sequence. Besides, most of search engines are based on keyword matching without considering the semantics of natural language queries [17]. It is often difficult to discover relevant code snippets and associated APIs.

Recently, Raghothaman et al. [36] proposed SWIM, which translates a natural language query to a list of possible APIs using a statistical word alignment model [12]. SWIM then uses the API list to retrieve relevant API sequences. However, the statistical word alignment model it utilizes is based on a bag-of-words assumption without considering the sequence of words and APIs. Therefore, it cannot recognize the deep semantics of a natural language query. For example, as described in their paper [36], it is difficult to distinguish the query *convert int to string* from *convert string to int*.

To address these issues, we propose DeepAPI, a novel, deep-learning based method that generates relevant API usage sequences given a natural language query. We formulate the API learning problem as a machine translation problem: given a natural language query $x = x_1, ..., x_N$ where x_i is a keyword, we aim to translate it into an API sequence

FSE'16, November 13–18, 2016, Seattle, WA, USA
© 2016 ACM. 978-1-4503-4218-6/16/11...$15.00
http://dx.doi.org/10.1145/2950290.2950334

$y = y_1, ..., y_T$ where y_j is an API. DeepAPI shows a deep understanding of natural language queries in two aspects:

- First, instead of matching keywords, DeepAPI learns the semantics of words by embedding them into a vector representation of context, so that semantically related words can be recognized.
- Second, instead of word-to-word alignment, DeepAPI learns the sequence of words in a natural language query and the sequence of associated APIs. It can distinguish the semantic differences between queries with different word sequences.

DeepAPI adapts a neural language model named RNN Encoder-Decoder [14]. Given a corpus of annotated API sequences, i.e., ⟨API sequence, annotation⟩ pairs, DeepAPI trains the language model that encodes each sequence of words (annotation) into a fixed-length context vector and decodes an API sequence based on the context vector. Then, in response to an API-related user query, it generates API sequences by consulting the neural language model.

To evaluate the effectiveness of DeepAPI, we collect a corpus of 7 million annotated code snippets from GitHub. We select 10 thousand instances for testing and the rest for training the model. After 240 hours of training (1 million iterations), we measure the accuracy of DeepAPI using BLEU score [35], a widely used accuracy measure for machine translation. Our results show that DeepAPI achieves an average BLEU score of 54.42, outperforming two related approaches, that is, code search with pattern mining (11.97) and SWIM [36] (19.90). We also ask DeepAPI 30 API-related queries collected from real query logs and related work. On average, the rank of the first relevant result is 1.6. 80% of the top 5 returned results and 78% of the top 10 returned results are deemed relevant. Our evaluation results confirm the effectiveness of DeepAPI.

The main contributions of our work are as follows:

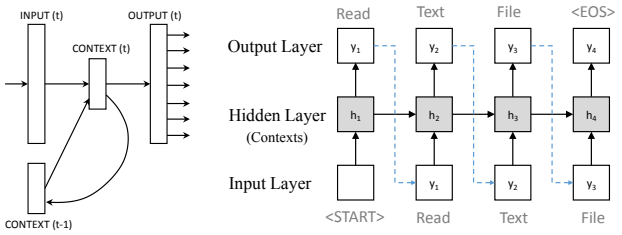
- To our knowledge, we are the first to adapt a deep learning technique to API learning. Our approach leads to more accurate API usage sequences as compared to the state-of-the-art techniques.
- We develop DeepAPI[1], a tool that generates API usage sequences based on natural language queries. We empirically evaluate DeepAPI's accuracy using a corpus of 7 million annotated Java code snippets.

The rest of this paper is organized as follows. Section 2 describes the background of the deep learning based neural language model. Section 3 describes the application of the RNN Encoder-Decoder, a deep learning based neural language model, to API learning. Section 4 describes the detailed design of our approach. Section 5 presents the evaluation results. Section 6 discusses our work, followed by Section 7 that presents the related work. We conclude the paper in Section 8.

2. DEEP LEARNING FOR SEQUENCE GENERATION

Our work adopts and augments recent advanced techniques from deep learning and neural machine translation [8, 14, 43]. These techniques are on the basis of Sequence-to-Sequence Learning [43], namely, generating a sequence (usually a natural language sentence) conditioned on another sequence. In this section, we discuss the background of these techniques.

[1]available at: http://www.cse.ust.hk/~xguaa/deepapi



(a) RNN Structure (b) RNNLM for sentence estimation

Figure 1: Illustration of the RNN Language Model

2.1 Language Model

It has been observed that software has naturalness [16]. Statistical language models have been adapted to many software engineering tasks [26] such as learning natural code conventions [5], code suggestion [44], and code completion [37]. These techniques regard source code as a special language and analyze it using statistical NLP techniques.

The language model is a probabilistic model of how to generate sentences in a language. It tells how likely a sentence would occur in a language. For a sentence y, where $y = (y_1, ..., y_T)$ is a sequence of words, the language model aims to estimate the joint probability of its words $Pr(y_1, ..., y_T)$. Since

$$Pr(y_1, ..., y_T) = \prod_{t=1}^{T} Pr(y_t | y_1, ..., y_{t-1}) \quad (1)$$

it is equivalent to estimate the probability of each word in y given its previous words, namely, what a word might be given its predecessing words.

As $Pr(y_t | y_1, ..., y_{t-1})$ is difficult to estimate, most applications use "n-gram models" [11] to approximate it, that is,

$$Pr(y_t | y_1, ..., y_{t-1}) \backsimeq Pr(y_t | y_{t-n+1}, ..., y_{t-1}) \quad (2)$$

where an n-gram is defined as n consecutive words. This approximation means that the next word y_t is conditioned only on the previous $n-1$ words.

2.2 Neural Language Model

The neural language model is a language model based on neural networks. Unlike the n-gram model which predicts a word based on a fixed number of predecessing words, a neural language model can predict a word by predecessing words with longer distances. It is also powerful to learn distributed representations of words, i.e, word vectors [30]. We adopt RNNLM [29], a language model based on a deep neural network, that is, Recurrent Neural Network (RNN) [29]. Figure 1a shows the basic structure of an RNN. The neural network includes three layers, that is, an input layer which maps each word to a vector, a recurrent hidden layer which recurrently computes and updates a hidden state after reading each word, and an output layer which estimates the probabilities of the following word given the current hidden state.

Figure 1b shows an example of how RNNLM estimates the probability of a sentence, that is, the probability of each word given predecessing words (Equation 1). To facilitate understanding, we expand the recurrent hidden layer for each individual time step. The RNNLM reads the words

in the sentence one by one, and predicts the possible following word at each time step. At step t, it estimates the probability of the following word $p(y_{t+1}|y_1, ..., y_t)$ by three steps: First, the current word y_t is mapped to a vector $\boldsymbol{y}_t$ by the input layer:

$$\boldsymbol{y}_t = input(y_t) \tag{3}$$

Then, it generates the hidden state (values in the hidden layer) $\boldsymbol{h}_t$ at time t according to the previous hidden state $\boldsymbol{h}_{t-1}$ and the current input $\boldsymbol{y}_t$:

$$\boldsymbol{h}_t = f(\boldsymbol{h}_{t-1}, \boldsymbol{y}_t) \tag{4}$$

Finally, the $Pr(y_{t+1}|y_1, ..., y_t)$ is predicted according to the current hidden state $\boldsymbol{h}_t$:

$$Pr(y_{t+1}|y_1, ..., y_t) = g(\boldsymbol{h}_t) \tag{5}$$

During training, the network parameters are learned from data to minimize the error rate of the estimated y (details are in [29]).

2.3 RNN Encoder-Decoder Model

The RNN Encoder-Decoder [14] is an extension of the basic neural language model (RNNLM). It assumes that there are two languages, a source language and a target language. It generates a sentence y of the target language given a sentence x of the source language. To do so, it first summarizes the sequence of source words $x_1, ..., x_{T_x}$ into a fixed-length context vector:

$$\boldsymbol{h}_t = f(\boldsymbol{h}_{t-1}, \boldsymbol{x}_t) \tag{6}$$

and

$$\boldsymbol{c} = \boldsymbol{h}_{T_x} \tag{7}$$

where f is a non-linear function that maps a word of source language $\boldsymbol{x}_t$ into a hidden state $\boldsymbol{h}_t$ at time t by considering the previous hidden state $\boldsymbol{h}_{t-1}$. The last hidden state $\boldsymbol{h}_{T_x}$ is selected as a context vector $\boldsymbol{c}$.

Then, it generates the target sentence y by sequentially predicting a word y_t conditioned on the source context $\boldsymbol{c}$ as well as previous words $y_1, ..., y_{t-1}$:

$$Pr(y) = \prod_{t=1}^{T} p(y_t|y_1, ..., y_{t-1}, \boldsymbol{c}) \tag{8}$$

The above procedures, i.e., f and p can be represented using two recurrent neural networks respectively, an encoder RNN which learns to transform a variable length source sequence into a fixed-length context vector, and a decoder RNN which learns a target language model and generates a sequence conditioned on the context vector. The encoder RNN reads the source words one by one. At each time stamp t, it reads one word, then updates and records a hidden state. When reading a word, it computes the current hidden state $\boldsymbol{h}_t$ using the current word $\boldsymbol{x}_t$ and the previous hidden state $\boldsymbol{h}_{t-1}$. When it finishes reading the end-of-sequence word *<EOS>*, it selects the last hidden state $\boldsymbol{h}_{T_x}$ as a context vector $\boldsymbol{c}$. The decoder RNN then sequentially generates the target words by consulting the context vector (Equation 8). It first sets the context vector as an initial hidden state of the decoder RNN. At each time stamp t, it generates one word based on the current hidden state and the context vector. Then, it updates the hidden state using the generated word (Equation 6). It stops when generating the end-of-sentence word *<EOS>*.

The RNN Encoder-Decoder model can then be trained to maximize the conditional log-likelihood [14], namely, minimize the following objective function:

$$L(\theta) = \frac{1}{N} \sum_{i=1}^{N} \sum_{t=1}^{T} \text{cost}_{it} \tag{9}$$

where N is the total number of training instances, while T is the length of each target sequence. $cost_{it}$ is the cost function for the t-th target word in instance i. It is defined as the negative log likelihood:

$$\text{cost}_{it} = -\log p_\theta(y_{it}|x_i) \tag{10}$$

where θ denotes model parameters such as weights in the neural network, while $p_\theta(y_{it}|x_i)$ (derived from Equation 6 to 8) denotes the likelihood of generating the t-th target word given the source sequence x_i in instance i according to the model parameters θ. Through optimizing the objective function using optimization algorithms such as gradient descendant, the optimum θ value can be estimated.

3. RNN ENCODER-DECODER MODEL FOR API LEARNING

3.1 Application of RNN Encoder-Decoder to API Learning

Now we present the idea of applying the RNN Encoder-Decoder model to API learning. We regard user queries as the source language and API sequences as the target language. Figure 2 shows an example of the RNN Encoder-Decoder model for translating a sequence of English words *read text file* to a sequence of APIs. The encoder RNN reads the source words one by one. When it reads the first word *read*, it embeds the word into vector $\boldsymbol{x}_1$ and computes the current hidden state $\boldsymbol{h}_1$ using $\boldsymbol{x}_1$. Then, it reads the second word *text*, embeds it into $\boldsymbol{x}_2$, and updates the hidden state $\boldsymbol{h}_1$ to $\boldsymbol{h}_2$ using $\boldsymbol{x}_2$. The procedure continues until the encoder reads the last word *file* and gets the final state $\boldsymbol{h}_3$. The final state $\boldsymbol{h}_3$ is selected as a context vector $\boldsymbol{c}$.

The decoder RNN tries to generate APIs sequentially using the context vector $\boldsymbol{c}$. It first generates *<START>* as the first word $\boldsymbol{y}_0$. Then, it computes a hidden state $\boldsymbol{h}_1$ based on the context vector $\boldsymbol{c}$ and $\boldsymbol{y}_0$, and predicts the first API *FileReader.new* according to $\boldsymbol{h}_1$. It then computes the next hidden state $\boldsymbol{h}_2$ according to the previous word vector $\boldsymbol{y}_1$, the context vector $\boldsymbol{c}$, and predicts the second API *BufferedReader.new* according to $\boldsymbol{h}_2$. This procedure continues until it predicts the end-of-sequence word *<EOS>*.

Different parts of a query could have different importance to an API in the target sequence. For example, considering the query *save file in default encoding* and the target API sequence *File.new FileOutputStream.new FileOutputStream.write FileOutputStream.close*, the word *file* is more important than *default* to the target API *File.new*. In our work, we adopt the attention-based RNN Encoder-Decoder model [8], which is a recent model that selects the important parts from the input sequence for each target word. Instead of generating target words using the same context vector $\boldsymbol{c}$ ($\boldsymbol{c} = \boldsymbol{h}_{T_x}$), an attention model defines individual $\boldsymbol{c}_j$'s for each target word y_j as a weighted sum of all historical hidden states $\boldsymbol{h}_1, ..., \boldsymbol{h}_{T_x}$. That is,

$$\boldsymbol{c}_j = \sum_{t=1}^{T_x} \alpha_{jt} \boldsymbol{h}_t \tag{11}$$

where each α_{jt} is a weight between the hidden state $\boldsymbol{h}_t$ and the target word y_j, while α can be modeled using another neural network and learned during training (see details in [8]).

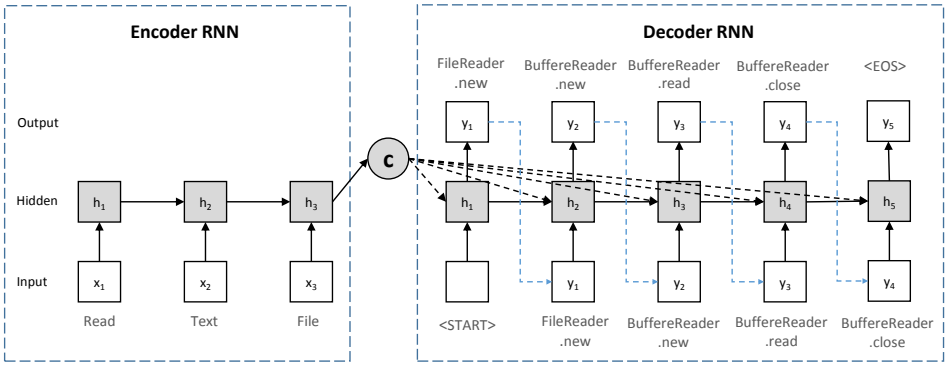


Figure 2: An Illustration of the RNN Encoder-Decoder Model for API learning

3.2 Enhancing RNN Encoder-Decoder Model with API importance

The basic RNN Encoder-Decoder model does not consider the importance of individual words in the target sequence either. In the context of API learning, different APIs have different importance for a programming task [27]. For example, the API *Logger.log* is widely used in many code snippets. However, it cannot help understand the key procedures of a programming task. Such ubiquitous APIs would be "weakened" during sequence generation.

We augment the RNN Encoder-Decoder model to predict API sequences by considering the individual importance of APIs. We define IDF-based weighting to measure API importance as follows:

$$w_{idf}(y_t) = log(\frac{N}{n_{y_t}}) \quad (12)$$

where N is the total number of API sequences in the training set and n_{y_t} denotes the number of sequences where the API y_t appears in the training set. Using IDF, the APIs that occur ubiquitously have the lower weights while the less common APIs have the higher weights.

We use API weight as a penalty term to the cost function (Equation 10). The new cost function of the RNN Encoder-Decoder model is:

$$\text{cost}_{it} = -\log p_\theta(y_{it}|x_i) - \lambda w_{idf}(y_t) \quad (13)$$

where λ denotes the penalty of IDF weight and is set empirically.

4. DEEPAPI: DEEP LEARNING FOR API SEQUENCE GENERATION

In this section, we describe DeepAPI, a deep-learning based method that generates relevant API usage sequences given an API-related natural language query. DeepAPI adapts the RNN Encoder-Decoder model for the task of API learning. Figure 3 shows the overall architecture of DeepAPI. It includes an offline training stage and an online translation stage. In the training stage, we prepare a large-scale corpus of annotated API sequences (API sequences with corresponding natural language annotations). The annotated API sequences are used to train a deep learning model, i.e., the RNN Encoder-Decoder language model as described in Section 3. Given an API-related user query, a ranked list of API sequences can be generated by the language model.

In theory our approach could generate APIs written in any programming languages. In this paper we limit our scope to the JDK library. The details of our method are explained in the following sections.

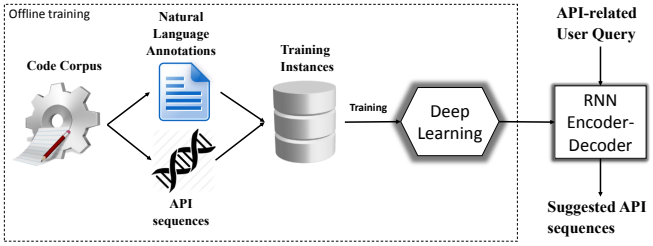


Figure 3: The Overall Workflow of DeepAPI

4.1 Gathering a Large-scale API Sequence to Annotation Corpus

We first construct a large-scale database that contains pairs of API sequences and natural language annotations for training the RNN Encoder-Decoder model. We download Java projects from GitHub [2] created from 2008 to 2014. To remove toy or experimental programs, we only select the projects with at least one star. In total, we collected 442,928 Java projects from GitHub. We use the last snapshot of each project. Having collected the code corpus, we extract ⟨API sequence, annotation⟩ pairs as follows:

4.1.1 Extracting API Usage Sequences

To extract API usage sequences from the code corpus, we parse source code files into ASTs (Abstract Syntax Trees) using Eclipse's JDT compiler [1]. The extraction algorithm starts from the dependency analysis of a whole project repository. We analyze all classes, recording field declarations together with their type bindings. We replace all object types with their real class types. Then, we extract API sequence from individual methods by traversing the AST of the method body:

- For each constructor invocation *new* $C()$, we append the API $C.new$ to the API sequence.
- For each method call $o.m()$ where o is an instance of a JDK class C, we append the API $C.m$ to the API sequence.
- For a method call passed as a parameter, we append the method before the calling method. For example, $o_1.m_1(o_2.m_2(),o_3.m_3())$, we produce a sequence $C_2.m_2$-$C_3.m_3$-$C_1.m_1$, where C_i is the JDK class of instance o_i.
- For a sequence of statements $stmt_1$; $stmt_2$;...; $stmt_t$, we extract the API sequence s_i from each statemen-

```
/***
 * Copies bytes from a large (over 2GB) InputStream to an OutputStream.
 * This method uses the provided buffer, so there is no need to use a
 * BufferedInputStream.
 * @param input the InputStream to read from
 *  . . .
 * @since 2.2
 */
public static long copyLarge(final InputStream input,
   final OutputStream  output, final byte[] buffer) throws IOException {
   long count = 0;
   int n;
   while (EOF != (n = input.read(buffer))) {
      output.write(buffer, 0, n);
      count += n;
   }
   return count;
}
```

API sequence: InputStream.read → OutputStream.write
Annotation: copies bytes from a large inputstream to an outputstream.

Figure 4: An example of extracting API sequence and its annotation from a Java method *IOUtils.copyLarge*[7]

t $stmt_i$, concatenate them, and produce the API sequence s_1-s_2-...-s_t.

- For conditional statements such as if ($stmt_1$) { $stmt_2$; } else { $stmt_3$; }, we create a sequence from all possible branches, that is, s_1-s_2-s_3, where s_i is the API sequence extracted from the statement $stmt_i$.
- For loop statements such as while($stmt_1$){$stmt_2$;}, we produce a sequence s_1-s_2, where s_1 and s_2 are API sequences extracted from the statement $stmt_1$ and $stmt_2$, respectively.

4.1.2 Extracting Annotations

To annotate the obtained API sequences with natural language descriptions, we extract method-level code summaries, specifically, the first sentence of a documentation comment[2] for a method. According to the Javadoc guidance[3], the first sentence is used as a short summary of a method. Figure 4 shows an example of documentation comment for a Java method *IOUtils.copyLarge*[4] in the Apache commons-io library.

We use the Eclipse JDT compiler for the extraction. For each method, we traverse its AST and extract the JavaDoc Comment part. We ignore methods without JavaDoc comments. Then, we select the first sentence of the comment as the annotation. We exclude irregular annotations such as those starting with "TODO: Auto-generated method stub", "NOTE:", and "test". We also filter out non-words and words within brackets in the annotations.

Finally, we obtain a database consisting of 7,519,907 ⟨API sequence, annotation⟩ pairs.

4.2 Training Encoder-Decoder Language Model

As described in Section 3, we adapt the attention-based RNN Encoder-Decoder model for API learning. The RNN has various implementations, we use GRU [14] which is a state-of-the-art RNN and performs well in many tasks [8, 14]. We construct the model as follows: we use two RNNs for the encoder - a forward RNN that directly encodes the source sentence and a backward RNN that encodes the reversed source sentence. Their output context vectors are concatenated to the decoder, which is also an RNN. All RNNs have 1000 hidden units. We set the dimension of word embedding to 120. We discuss the details of parameter tuning in Section 5.4.

All models are trained using the minibatch stochastic gradient descent algorithm(SGD) [9, 22] together with Adadelta [51], which automatically adjusts the learning rate. We set the batch size (i.e., number of instances per batch) as 200. For training the neural networks, we limit the source and target vocabulary to the top 10,000 words that are most frequently used in API sequences and annotations.

For implementation, we use GroundHog [8, 14], an open-source deep learning framework. We train our models in a server with one Nvidia K20 GPU. The training lasts ∼240 hours with 1 million iterations.

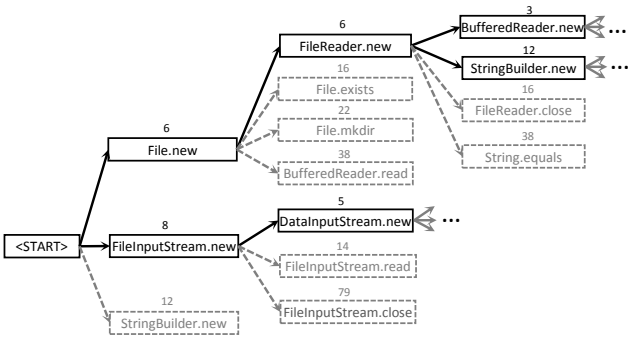


Figure 5: An illustration of beam search (beam width=2)

4.3 Translation

So far we have discussed the training of a neural language model, which outputs the most likely API sequence given a natural language query. However, an API could have multiple usages. To obtain a ranked list of possible API sequences for user selection, we need to generate more API sequences according to their probability at each step.

DeepAPI uses Beam Search [21], a heuristic search strategy, to find API sequences that have the least cost value(computed using Equation 13) given by the language model. Beam search searches APIs produced at each step one by one. At each time step, it selects n APIs from all branches with the least cost values, where n is the beam-width. It then prunes off the remaining branches and continues selecting the possible APIs that follow on until it meets the end-of-sequence symbol. Figure 5 shows an example of a beam search (beam-width=2) for generating an API sequence for the query "read text file". First, 'START' is selected as the first API in the generated sequence. Then, it estimates the probabilities of all possible APIs that follow on according to the language model. It computes their cost values according to Equation 13, and selects *File.new* and *FileInputStream.new* which have the least cost values of 6 and 8, respectively. Then, it ignores branches of other APIs and continue estimating possible APIs after *File.new* and *FileInputStream.new*. Once it selects an end-of-sequence symbol as the next API, it stops that branch and the branch is selected as a generated sequence.

Finally, DeepAPI produces n API sequences for each query where n is the beam-width. We rank the generated

[2]A documentation comment in JAVA starts with slash-asterisk-asterisk (/**) and ends with asterisk-slash (*/)
[3]http://www.oracle.com/technetwork/articles/java/index-137868.html
[4]https://github.com/apache/commons-io/blob/trunk/src/main/java/org/apache/commons/io/IOUtils.java

API sequences according to their average cost values during the beam search procedure.

5. EVALUATION

We evaluate the effectiveness of DeepAPI by measuring its accuracy on API sequence generation. Specifically, our evaluation addresses the following research questions:

- **RQ1: How accurate is DeepAPI for generating API usage sequences?**
- **RQ2: How accurate is DeepAPI under different parameter settings?**
- **RQ3: Do the enhanced RNN Encoder-Decoder models improve the accuracy of DeepAPI?**

5.1 Accuracy Measure

5.1.1 Intrinsic Measure - BLEU

We use the BLEU score [35] to measure the accuracy of generated API sequences. The BLEU score measures how close a candidate sequence is to a reference sequence (usually a human written sequence). It is a widely used accuracy measure for machine translation in the machine learning and natural language processing literature [8, 14, 43]. In our API learning context, we regard a generated API sequence given a query as a candidate, and a human-written API sequence (extracted from code) for the same query as a reference. We use BLEU to measure how close the generated API sequence is to a human-written API sequence.

Generally, BLEU measures the hits of n-grams of a candidate sequence to the reference. It is computed as:

$$BLEU = BP \cdot \exp(\sum_{n=1}^{N} w_n \log p_n) \tag{14}$$

where each p_n is the precision of n-grams, that is, the ratio of length n subsequences in the candidate that are also in the reference:

$$p_n = \frac{\text{\# n-grams appear in the reference+1}}{\text{\# n-grams of candidate+1}} \quad \text{for } n = 1, ..., N \tag{15}$$

where N is the maximum number of grams we consider. We set N to 4, which is a common practice in the Machine Learning literature [43]. Each w_n is the weight of each p_n. A common practice is to set $w_n = \frac{1}{N}$. BP is a brevity penalty which penalties overly short candidates (that may have a higher n-gram precision).

$$BP = \begin{cases} 1 & \text{if } c > r \\ e^{(1-r/c)} & \text{if } c \leq r \end{cases} \tag{16}$$

where r is the length of the reference sequence, and c is the length of the candidate sequence.

We now give an example of BLEU calculation. For a candidate API sequence {a-c-d-b} and a reference API sequence {a-b-c-d-e}, their 1-grams are {a,b,c,d} and {a,b,c,d,e}. All four 1-grams of the candidate are hit in the reference. Then, $p_1 = \frac{4+1}{4+1} = 1$. Their 2-grams are {ac,cd,db} and {ab,bc,cd,de}, respectively. Then, $p_2 = \frac{1+1}{3+1} = \frac{1}{2}$ as only cd is matched. $p_3 = \frac{0+1}{2+1} = \frac{1}{3}$ and $p_4 = \frac{0+1}{1+1} = \frac{1}{2}$ as no 3-gram nor 4-gram is matched. As their lengths are 4 and 5 respectively, $BP = e^{(1-5/4)} = 0.78$. The final BLEU is $0.78 \times exp(\frac{1}{4} \times log1 + \frac{1}{4} \times log\frac{1}{2} + \frac{1}{4} \times log\frac{1}{3} + \frac{1}{4} \times log\frac{1}{2}) = 41.91\%$

BLEU is usually expressed as a percentage value between 0 and 100. The higher the BLEU, the closer the candidate sequence is to the reference. If the candidate sequence is completely equal to the reference, the BLEU becomes 100%.

5.1.2 Extrinsic Measures - FRank and Relevancy Ratio

We also use two measures for human evaluation. They are FRank and relevancy ratio [36]. FRank is the rank of the first relevant result in the result list [36]. It is important as most users scan the results from top to bottom.

The relevancy ratio is defined as the precision of relevant results in a number of results [36].

$$\text{relevancy ratio} = \frac{\text{\#relevant results}}{\text{\#all selected results}} \tag{17}$$

The value of both measures ranges from 0 to 100. The higher the better.

5.2 Comparison Methods

We compare the accuracy of our approach with that of two state-of-the-art API learning approaches, namely Code Search with Pattern Mining [24, 46] and SWIM [36].

5.2.1 Code Search with Pattern Mining

To obtain relevant API sequences for a given a query, one can perform code search over the code corpus using information retrieval techniques [20, 24, 25, 28], and then utilize an API usage pattern miner [15, 46, 49] to identify an appropriate API sequences in the returned code snippets.

We compare DeepAPI with this approach. We use Lucene [4] to perform a code search for a given natural language query and UP-Miner [46] to perform API usage pattern mining. Lucene is an open-source information retrieval engine, which has been integrated into many code search engines [24, 36]. Much the same as these code search engines do, we treat source code as plain text documents and use Lucene to build source code index and perform text retrieval. UP-Miner [46] is a pattern mining tool, which produces API sequence patterns from code snippets. It first clusters API sequences extracted from code snippets, and then identifies frequent patterns from the clustered sequences. Finally, it clusters the frequent patterns to reduce redundancy. We use UP-Miner to mine API usage sequences from the code snippets returned by the Lucene-based code search.

In this experiment, we use the same code corpus as used for evaluating DeepAPI, and compare the BLEU scores with those of DeepAPI.

5.2.2 SWIM

SWIM [36] is a recently proposed code synthesis tool, which also supports API sequence search based on a natural language query. Given a query, it expands the query keywords to a list of relevant APIs using a statistical word alignment model [12]. With the list of possible APIs, SWIM searches related API sequences using Lucene [4]. Finally, it synthesizes code snippets based on the API sequences. As code synthesis is beyond our scope, we only compare DeepAPI with the API learning component of SWIM, that is, from a natural language query to an API sequence. In their experiments, SWIM uses Bing clickthrough data to build the model. In our experiment, for fair comparison, we evaluate SWIM using the same dataset as we did for evaluating DeepAPI. That is, we train the word alignment model and

Table 1: BLEU scores of DeepAPI and related techniques (%)

Tool	Top1	Top5	Top10
Lucene+UP-Miner	11.97	24.08	29.64
SWIM	19.90	25.98	28.85
DeepAPI	54.42	64.89	67.83

build API index on the training set, and evaluate the search results on the test set.

5.3 Accuracy (RQ1)

5.3.1 Intrinsic Evaluation

Evaluation Setup: We first evaluate the accuracy of generated API sequences using the BLEU score. As described in Section 4.1, we collect a database comprising 7,519,907 ⟨API sequence, annotation⟩ pairs. We split them into a test set and a training set. The test set comprises 10,000 pairs while the training set consists of the remaining instances. We train all models using the training set and compute the BLEU scores in the test set. We calculate the highest BLEU score for each test instance in the top n results.

Results: Table 1 shows the BLEU scores of DeepAPI, SWIM, and Code Search (Lucene+UP-Miner). Each column shows the average BLEU score for a method. As the results indicate, DeepAPI produces API sequences with higher accuracy. When only the top 1 result is examined, the BLEU score achieved by DeepAPI is 54.42, which is greater than that of SWIM (BLEU=19.90) and Code Search (BLEU=11.97). The improvement over SWIM is 173% and the improvement over Code Search is 355%. Similar results are obtained when the top 5 and 10 results are examined. The evaluation results confirm the effectiveness of the deep learning method used by DeepAPI.

5.3.2 Extrinsic Evaluation

To further evaluate the relevancy of the results returned by DeepAPI, we selected 17 queries used in [36]. These queries have corresponding Java APIs and are commonly occurring queries in the Bing search log [36]. To demonstrate the advantages of DeepAPI, we also designed 13 longer queries and queries with semantically similar words. In total, 30 queries are used. These queries do not appear in the training set. Table 2 lists the queries.

For each query, the top 10 returned results by DeepAPI and SWIM are manually examined. To reduce labeling bias, two developers separately label the relevancy of each resulting sequence and combine their labels. For inconsistent labels, they discuss and relabel them until a settlement is reached. The FRank and the relevancy ratios for the top 5 and top 10 returned results are then computed. To test the statistical significance, we apply the Wilcoxon signed-rank test (p<0.05) for all results of both approaches. A resulting p-value less than 0.05 indicates that the differences between DeepAPI and SWIM are statistically significant.

Table 2 shows the accuracy of both DeepAPI and SWIM. The symbol '-' means no relevant result has been returned within the top 10 results. The results show that DeepAPI is able to produce mostly relevant results. It achieves an average FRank of 1.6, an average top 5 accuracy of 80%, and an average top 10 accuracy of 78%. Furthermore, DeepAPI produces more relevant API sequences than SWIM, whose average top 5 and top 10 accuracy is 44% and 47%, respectively. For some queries, SWIM failed to obtain relevant results in the top 10 returned results. We conservatively treat the FRank as 11 for these unsuccessful queries. Then, the FRank achieved by SWIM is greater than 4.0, which is much higher than what DeepAPI achieved (1.60). The p-values for the three comparisons are 0.01, 0.02 and 0.01, respectively, indicating statistical significance of the improvement of DeepAPI over SWIM. In summary, the evaluation results confirm the effectiveness of DeepAPI.

Table 2 also shows examples of generated sequences by DeepAPI. We can see that DeepAPI is able to distinguish word sequences. For example, DeepAPI successfully distinguishes the query *convert int to string* from *convert string to int*. Another successful example is the query expansion. For example, the query *save an image to a file* and *write an image to a file* return similar results. DeepAPI also performs well in longer queries such as *copy a file and save it to your destination path* and *play the audio clip at the specified absolute URL*. Such queries comprise many keywords, and DeepAPI can successfully recognize the semantics.

We also manually check the results returned by SWIM. We find SWIM may return partially matched sequences. For example, for the query *generate md5 hash code*, SWIM returns many results containing only *Object.hashCode*, which simply returns a hash code. SWIM also returns project specific results without fully understanding the query. For example, for the query "test file exists", SWIM returns "*File.new, File.exists, File.getName, File.new, File.delete, FileInputStream.new, FileInputStream.read, ...*", which is not only related to file existence test, but also to other project-specific tasks. Such project specific results can also be seen for the query *create file*. Compared with DeepAPI, SWIM performs worse in long queries. For example, SWIM performs worse in the query *copy a file and save it to your destination path* than in the query *copy file*. This is because long queries often have multiple objectives, which cannot be understood by SWIM.

Still, DeepAPI could return inaccurate or partial results. For example, for the query *parse xml*, it returns related APIs *InputSource.new, DocumentBuilder.parse*. But it misses the APIs about how *DocumentBuilder* is created (*DocumentBuilderFactory.newDocumentBuilder*). The reason could be that an API sequence may be called in an inter-procedural manner. When preparing the training set, we only consider API sequences within one method. The API *DocumentBuilderFactory.newDocumentBuilder* could be called in another method and is passed as a parameter. This causes incomplete sequences in the training set. In the future, we will perform more accurate program analysis and create a better training set.

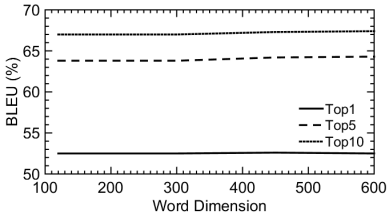
5.4 Accuracy Under Different Parameter Settings (RQ2)

We also qualitatively compare the accuracy of DeepAPI in different parameter settings. We analyze two parameters, that is, the dimension of word embedding and the number of hidden units. We vary the values of these two parameters and evaluate their impact on the BLEU scores.

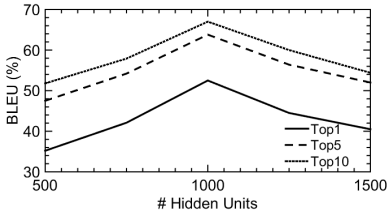
Figure 6 shows the influence of different parameter settings on the test set. The dimension of word embedding makes little difference to the accuracy. The accuracy of DeepAPI greatly depends on the number of hidden units

Table 2: Queries for Human Evaluation (FR: FRank, RR5: top 5 relevancy ratio, RR10: top 10 relevancy ratio)

query (How to...)	SWIM			DeepAPI			Generated API sequence by DeepAPI
	FR	RR5	RR10	FR	RR5	RR10	
convert int to string	8	0	10	2	40	90	Integer.toString
convert string to int	1	80	80	1	100	100	Integer.parseInt String.toCharArray Character.digit
append strings	3	60	80	1	100	100	StringBuilder.append StringBuilder.toString
get current time	1	80	80	10	10	10	System.currentTimeMillis Timestamp.new
parse datetime from string	9	0	10	1	100	80	SimpleDateFormat.new SimpleDateFormat.parse
test file exists	-	0	0	1	100	100	File.new File.exists
open a url	1	100	100	1	100	100	URL.new URL.openConnection
open file dialog	-	0	0	1	100	80	JFileChooser.new JFileChooser.showOpenDialog JFileChooser.getSelectedFile
get files in folder	2	40	20	3	40	50	File.new File.list File.new File.isDirectory
match regular expressions	1	100	100	1	80	90	Pattern.compile Pattern.matcher Matcher.group
generate md5 hash code	1	60	40	1	100	100	MessageDigest.getInstance MessageDigest.update MessageDigest.digest
generate random number	7	0	10	1	100	70	Random.new Random.nextInt
round a decimal value	-	0	0	1	100	100	Math.floor Math.pow Math.round
execute sql statement	2	80	80	1	80	60	Connection.prepareStatement PreparedStatement.execute PreparedStatement.close
connect to database	7	0	20	1	100	90	Properties.getProperty Class.forName DriverManager.getConnection
create file	10	0	10	3	40	20	File.exists File.createNewFile
copy file	1	100	100	2	20	10	FileInputStream.new FileOutputStream.new FileInputStream.read FileOutputStrem.write FileInputStream.close FileOutputStream.close
copy a file and save it to your destination path	1	20	50	1	100	100	FileInputStream.new FileOutputStream.new FileInputStream.getChannel FileOutputStream.getChannel FileChannel.size FileChannel.transferTo FileInputStream.close FileOutputStream.close FileChannel.close FileChannel.close
delete files and folders in a directory	1	100	90	1	100	100	File.isDirectory File.list File.new File.delete
reverse a string	3	20	10	2	60	70	StringBuffer.new StringBuffer.reverse
create socket	-	0	0	1	60	80	ServerSocket.new ServerSocket.bind
rename a file	-	0	0	1	100	100	File.renameTo File.delete
download file from url	2	60	80	1	100	80	URL.new URL.openConnection URLConnection.getInputStream BufferedInputStream.new
serialize an object	1	100	100	3	60	70	ObjectOutputStream.new ObjectOutputStream.writeObject ObjectOutputStream.close
read binary file	4	40	70	1	100	80	DataInputStream.new DataInputStream.readInt DataInputStream.close
save an image to a file	1	20	10	1	80	80	File.new ImageIO.write
write an image to a file	1	20	10	1	100	90	File.new ImageIO.write
parse xml	1	100	100	1	80	60	InputSource.new DocumentBuilder.parse
play audio	1	100	100	1	60	80	SourceDataLine.open SourceDataLine.start
play the audio clip at the specified absolute URL	1	40	50	1	100	90	Applet.getAudioClip AudioClip.play
average	>4.0	44	47	1.6	80	78	



(a) Performance of different dimensions of word embedding



(b) Performance of different numbers of hidden units

Figure 6: BLEU scores of different parameter settings

in the hidden layer. The optimum number of hidden units is around 1000.

5.5 Performance of the Enhanced RNN Encoder-Decoder Models (RQ3)

In Section 3, we describe two enhancements to the original RNN Encoder-Decoder model, for the task of API learning: an attention-based RNN Encoder-Decoder proposed by [8] (Section 3.1) and an enhanced RNN Encoder-Decoder with a new cost function (Section 3.2) proposed by us. We now evaluate if the enhanced models improve the accuracy of DeepAPI when constructed using the original RNN Encoder-Decoder model.

Table 3: BLEU scores of different RNN Encoder-Decoder Models (%)

Encoder-Decoder Model	Top1	Top5	Top10
RNN	48.83	60.98	64.27
RNN+Attention	52.49	63.81	66.97
RNN+Attention+New Cost Function	54.42	64.89	67.83

Table 3 shows the BLEU scores of the three models. The attention-based RNN Encoder-Decoder outperforms the basic RNN Encoder-Decoder model on API learning. The relative improvement in the top 1, 5, and 10 results (in terms of BLEU score) is 8%, 5% and 4%, respectively. This result confirms the effectiveness of the attention-based RNN Encoder-Decoder used in our approach.

Table 3 also shows that the enhanced model with the new cost function leads to better results as compared to the attention-based RNN Encoder-Decoder model. The improvement in the top 1, 5, and 10 results (in terms of BLEU score) is 4%, 2% and 1%, respectively. Figure 7 shows that the performance of the enhanced model are slightly different under different parameter settings, with an optimum λ of around 0.035. The results confirm the usefulness of the proposed cost function for enhancing the RNN Encoder-Decoder model.

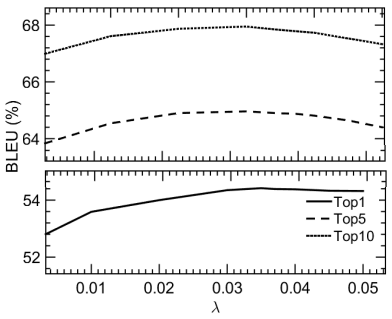


Figure 7: Performance of the Enhanced RNN Encoder-Decoder Model under Different Settings of λ

6. DISCUSSION

6.1 Why does DeepAPI work?

A major challenge for API learning is the semantic gap between code and natural language descriptions. Existing information retrieval based approaches usually have a bag-of-words assumption and lack a deep understanding of the high-level semantics of natural language and code. We have identified three advantages of DeepAPI that address this problem.

Word embedding and query expansion A significant difference between DeepAPI and bag-of-words methods is, DeepAPI embeds words into a continuous semantic space where the semantically similar words are placed close to each other. When reading words in a query, the model maps them to semantic vectors. Words with similar semantics have similar vector representations and have a similar impact on the hidden states of the RNN encoder. Therefore, queries with semantically similar words can lead to similar results. Figure 8 shows a 2-D projection of the encoded vectors of queries. These queries are selected from the 10,000 annotations in the test set. For ease of demonstration, we select queries with a keyword "file" and exclude those longer than eight words. As shown in the graph, DeepAPI can successfully embed similar queries into a nearby place. There are three clear clusters of queries, corresponding to "read/load files","write/save files", and "remove/delete files". Queries with semantically related words are close to each other. For example, queries starting with *save*, *write*, and *output* are in the same "cluster" though they contain different words.

Learning sequence instead of bag-of-words The hidden layer of the encoder has the memory capacity. It considers not only the individual words, but also their relative positions. Even for the same word set, different sequences will be encoded to different vectors, resulting in different API sequences. In that sense, DeepAPI learns not only the words, but also phrases. While traditional models simply consider individual words or word-level alignments. A typical example is that, queries with different word sequences such as *convert int to string* and *convert string to int* can be distinguished well by DeepAPI.

Generating common patterns instead of searching specific samples Another advantage of our approach is that, it can learn common patterns of API sequences. The decoder itself is a language model and remembers the likelihoods of different sequences. Those common sequences will have high probabilities according to the model. Therefore, it tends to generate common API sequences rather than project-specific ones. On the other hand, the information retrieval based approaches simply consider searching individual instances and could return project-specific API sequences.

Though several techniques such as query expansion [18, 40, 50] and frequent pattern mining [49] can partially solve some of the above problems, their effectiveness remains to be improved. For example, it has been observed that expanding a code search query with inappropriate English synonyms can return even worse results as compared to the original query [41]. Furthermore, few techniques can exhibit all the above advantages.

6.2 Threats to Validity

We have identified the following threats to validity:

All APIs studied are Java APIs All APIs and related projects investigated in this paper are JDK APIs. Hence, they might not be representative of APIs for other libraries and programming languages. In the future, we will extend the model to other libraries and programming languages.

Quality of annotations We collected annotations of API sequences from the first sentence of documentation comments. Other sentences in the comments may also be informative. In addition, the first sentences may have noise. In the future, we will investigate a better NLP technique to extract annotations for code.

Training dataset In the original SWIM paper [36], the clickthrough data from Bing.com is used for evaluation. Such data is not easy accessible for most researchers. For fair and easy comparison, we evaluate SWIM on the dataset collected from GitHub and Java documentations (the same for evaluating DeepAPI). We train the models using annotations of API sequences collected from the documentation comments. In the future, we will evaluate both SWIM and DeepAPI on a variety of datasets including the Bing clickthrough data. In the future, we will perform more accurate program analysis and create a better training set.

7. RELATED WORK

7.1 Code Search

There is a large amount of work on code search [10, 13, 17, 19, 25, 28]. For example, McMillan et al. [28] proposed a code search tool called Portfolio that retrieves and visualizes relevant functions and their usages. Chan and Cheng [13] designed an approach to help users find usages of APIs given only simple text phrases. Lv et al. [25] proposed CodeHow, a code search tool that incorporates an extended Boolean model and API matching. They first find relevant APIs to a query by matching the query to API documentation. Then, they improve code search performance by considering the APIs which are relevant to the query in code retrieval. As described in Section 6, DeepAPI differs from code search techniques in that it does not rely on information retrieval techniques and can understand word sequences and query semantics.

7.2 Mining API Usage Patterns

Instead of generating API sequences from natural language queries, there is a number of techniques focusing on mining API usage patterns [15, 31, 46, 49]. API usage patterns are frequent API method call sequences. Xie et al. [49] proposed MAPO, which is one of the first works on mining API patterns from code corpus. MAPO represents source

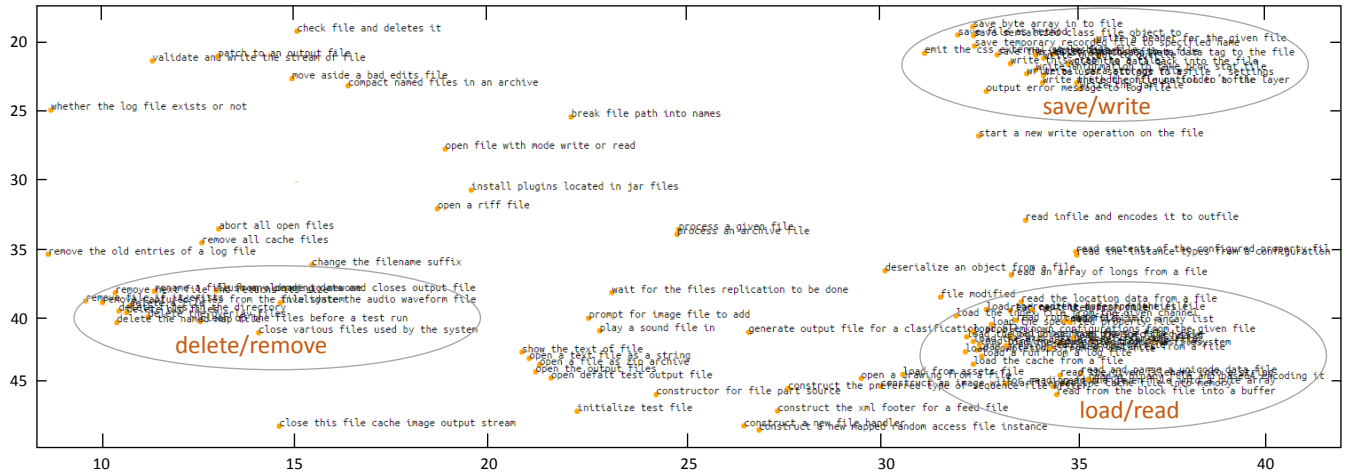


Figure 8: A 2D projection of embeddings of queries using t-SNE[45]

code as call sequences and clusters them according to similarity heuristics such as method names. It finally generates patterns by mining and ranking frequent sequences in each cluster. UP-Miner [46] is an improvement of MAPO, which removes the redundancy among patterns by two rounds of clustering of the method call sequences. By applying API usage pattern mining on large-scale code search results, these techniques can also return API usage sequences in response to user's natural language queries.

While the above techniques are useful for understanding the usage of an API, they are insufficient for answering the question of *which APIs to use*, which is the aim of DeepAPI. Furthermore, different from a frequent pattern mining approach, DeepAPI constructs a neural language model to learn usage patterns.

7.3 From Natural Language to Code

A number of related techniques have been proposed to generate code snippets from natural language queries. For example, Raghothaman et al. [36] proposed SWIM, a code synthesis technique that translates user queries into the APIs of interest using Bing search logs and then synthesizes idiomatic code describing the use of these APIs. SWIM has a component that produces API sequences given user's natural language query. Our approach and SWIM differ in many aspects. First, SWIM generates bags of APIs using statistical word alignment [12]. The word alignment model does not consider word embeddings and word sequences of natural language queries, and has limitations in query understanding. Second, to produce API sequences, SWIM searches API sequences from the code repository using a bag of APIs. It does not consider the relative position of different APIs. Fowkes and Sutton [7] build probabilistic models that jointly model short natural language utterances and source code snippets. The main differences between our approach and theirs are two-fold. First, they use a bag-of-words model to represent natural language sentences which will not recognize word sequences. Second, they use a traditional probabilistic model which is unable to recognize semantically related words.

7.4 Deep Learning for Source Code

Recently, some researchers have explored the possibility of applying deep learning techniques to source code [6, 32, 33, 37]. A typical application that leverages deep learning is to extract source code features [32, 47]. For example, Mou et al. [32] proposed to learn vector representations of source code for deep learning tasks. Mou et al. [33] also proposed convolutional neural networks over tree structures for programming language processing. Deep learning has also been applied to code generation [23, 34]. For example, Mou et al. [34] proposed to generate code from natural language user intentions using an RNN Encoder-Decoder model. Their results show the feasibility of applying deep learning techniques to code generation from a highly homogeneous dataset (simple programming assignments). Deep Learning has also been applied to code completion [37, 48]. For example, White et al. [48] applied the RNN language model to source code files and showed its effectiveness in predicting software tokens. Raychev et al. [37] proposed to apply the RNN language model to complete partial programs with holes. In our work, we explore the application of deep learning techniques to API learning.

8. CONCLUSION

In this paper, we apply a deep learning approach, RNN Encoder-Decoder, for generating API usage sequences for a given API-related natural language query. Our empirical study has shown that the proposed approach is effective in API sequence generation. Although deep learning has shown promise in other areas, we are the first to observe its effectiveness in API learning.

The RNN Encoder-Decoder based neural language model described in this paper may benefit other software engineering problems such as code search and bug localization. In the future, we will explore the applications of this model to these problems. We will also investigate the synthesis of sample code from the generated API sequences.

An online demo of DeepAPI can be found on our website at: http://www.cse.ust.hk/~xguaa/deepapi.

9. ACKNOWLEDGMENTS

We thank all the anonymous reviewers for their valuable comments. This work was conducted when the first author was a research intern at Microsoft Research.

10. REFERENCES

[1] Eclipse JDT. http://www.eclipse.org/jdt/.
[2] Github. https://github.com.

[3] Github search. https://github.com/search?type=code.

[4] Lucene. https://lucene.apache.org/.

[5] 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 (FSE'14)*, pages 281–293. ACM, 2014.

[6] M. Allamanis, H. Peng, and C. Sutton. A convolutional attention network for extreme summarization of source code. In *Proceedings of the International Conference on Machine Learning (ICML'16)*, 2016.

[7] M. Allamanis, D. Tarlow, A. Gordon, and Y. Wei. Bimodal modelling of source code and natural language. In *Proceedings of the 32nd International Conference on Machine Learning (ICML'15)*, pages 2123–2132, 2015.

[8] D. Bahdanau, K. Cho, and Y. Bengio. Neural machine translation by jointly learning to align and translate. *arXiv preprint arXiv:1409.0473*, 2014.

[9] L. Bottou. Large-scale machine learning with stochastic gradient descent. In *Proceedings of the 19th International Conference on Computational Statistics (COMPSTAT'10)*, pages 177–186. Springer, 2010.

[10] J. Brandt, M. Dontcheva, M. Weskamp, and S. R. Klemmer. Example-centric programming: integrating web search into the development environment. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, CHI'10, pages 513–522. ACM, 2010.

[11] P. F. Brown, P. V. Desouza, R. L. Mercer, V. J. D. Pietra, and J. C. Lai. Class-based n-gram models of natural language. *Computational linguistics*, 18(4):467–479, 1992.

[12] P. F. Brown, V. J. D. Pietra, S. A. D. Pietra, and R. L. Mercer. The mathematics of statistical machine translation: Parameter estimation. *Computational linguistics*, 19(2):263–311, 1993.

[13] W.-K. Chan, H. Cheng, and D. Lo. Searching connected API subgraph via text phrases. In *Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering*, FSE'12, pages 10:1–10:11. ACM, 2012.

[14] K. Cho, B. Van Merriënboer, Ç. Gülçehre, D. Bahdanau, F. Bougares, H. Schwenk, and Y. 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'14)*, pages 1724–1734, Doha, Qatar, Oct. 2014. Association for Computational Linguistics.

[15] J. Fowkes and C. Sutton. Parameter-free probabilistic API mining at github scale. In *Proceedings of the ACM SIGSOFT 24th International Symposium on the Foundations of Software Engineering (FSE'16)*. ACM, 2016.

[16] A. Hindle, E. T. Barr, Z. Su, M. Gabel, and P. Devanbu. On the naturalness of software. In *Proceedings of the 34th International Conference on Software Engineering (ICSE'12)*, pages 837–847. IEEE, 2012.

[17] R. Holmes, R. Cottrell, R. J. Walker, and J. Denzinger. The end-to-end use of source code examples: An exploratory study. In *Proceedings of the IEEE International Conference on Software Maintenance (ICSM'09)*, pages 555–558. IEEE, 2009.

[18] M. J. Howard, S. Gupta, L. Pollock, and K. Vijay-Shanker. Automatically mining software-based, semantically-similar words from comment-code mappings. In *Proceedings of the 10th Working Conference on Mining Software Repositories (MSR'13)*, pages 377–386. IEEE Press, 2013.

[19] I. Keivanloo, J. Rilling, and Y. Zou. Spotting working code examples. In *Proceedings of the 36th International Conference on Software Engineering (ICSE'14)*, pages 664–675. ACM, 2014.

[20] J. Kim, S. Lee, S. Hwang, and S. Kim. Towards an intelligent code search engine. In *Proceedings of the 24th AAAI Conference on Artificial Intelligence (AAAI'10)*, pages 1358–1363, 2010.

[21] P. Koehn. Pharaoh: a beam search decoder for phrase-based statistical machine translation models. In *Machine translation: From real users to research*, pages 115–124. Springer, 2004.

[22] M. Li, T. Zhang, Y. Chen, and A. J. Smola. Efficient mini-batch training for stochastic optimization. In *Proceedings of the 20th ACM SIGKDD international conference on Knowledge discovery and data mining (KDD'14)*, pages 661–670. ACM, 2014.

[23] W. Ling, E. Grefenstette, K. M. Hermann, T. Kocisky, A. Senior, F. Wang, and P. Blunsom. Latent predictor networks for code generation. *arXiv preprint arXiv:1603.06744*, 2016.

[24] E. Linstead, S. Bajracharya, T. Ngo, P. Rigor, C. Lopes, and P. Baldi. Sourcerer: mining and searching internet-scale software repositories. *Data Mining and Knowledge Discovery*, 18:300–336, 2009.

[25] F. Lv, H. Zhang, J. Lou, S. Wang, D. Zhang, and J. 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'15)*, pages 260–270. IEEE, 2015.

[26] C. Maddison and D. Tarlow. Structured generative models of natural source code. In *Proceedings of the 31st International Conference on Machine Learning (ICML'14)*, pages 649–657, 2014.

[27] C. McMillan, M. Grechanik, and D. Poshyvanyk. Detecting similar software applications. In *Proceedings of the 34th International Conference on Software Engineering (ICSE'12)*, pages 364–374. IEEE, 2012.

[28] C. McMillan, M. Grechanik, D. Poshyvanyk, Q. Xie, and C. Fu. Portfolio: finding relevant functions and their usage. In *Proceedings of the 33rd International Conference on Software Engineering (ICSE'11)*, pages 111–120. IEEE, 2011.

[29] T. Mikolov, M. Karafiát, L. Burget, J. Cernockỳ, and S. Khudanpur. Recurrent neural network based language model. In *Proceedings of the 11th Annual Conference of the International Speech Communication Association (INTERSPEECH'10)*, pages 1045–1048, 2010.

[30] 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 (NIPS'13)*, pages 3111–3119, 2013.

[31] E. Moritz, M. Linares-Vásquez, D. Poshyvanyk, M. Grechanik, C. McMillan, and M. Gethers. Export: Detecting and visualizing API usages in large source code repositories. In *Proceedings of the IEEE/ACM 28th International Conference on Automated Software Engineering (ASE'13)*, pages 646–651. IEEE, 2013.

[32] L. Mou, G. Li, Y. Liu, H. Peng, Z. Jin, Y. Xu, and L. Zhang. Building program vector representations for deep learning. *arXiv preprint arXiv:1409.3358*, 2014.

[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 (AAAI'16)*, 2016.

[34] L. Mou, R. Men, G. Li, L. Zhang, and Z. Jin. On end-to-end program generation from user intention by deep neural networks. *arXiv*, 2015.

[35] K. Papineni, S. Roukos, T. Ward, and W.-J. Zhu. BLEU: a method for automatic evaluation of machine translation. In *Proceedings of the 40th annual meeting on association for computational linguistics (ACL'02)*, pages 311–318. Association for Computational Linguistics, 2002.

[36] 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 (ICSE'16)*, pages 357–367. ACM, 2016.

[37] V. Raychev, M. Vechev, and E. Yahav. Code completion with statistical language models. In *ACM SIGPLAN Notices*, volume 49, pages 419–428. ACM, 2014.

[38] M. P. Robillard. What makes APIs hard to learn? answers from developers. *IEEE Software*, 26(6):27–34, 2009.

[39] M. P. Robillard and R. DeLine. A field study of API learning obstacles. *Empirical Software Engineering*, 16(6):703–732, 2010.

[40] D. Shepherd, Z. P. Fry, E. Hill, L. Pollock, and K. Vijay-Shanker. Using natural language program analysis to locate and understand action-oriented concerns. In *Proceedings of the 6th international conference on Aspect-oriented software development*, pages 212–224. ACM, 2007.

[41] G. Sridhara, E. Hill, L. Pollock, and K. Vijay-Shanker. Identifying word relations in software: A comparative study of semantic similarity tools. In *Proceedings of the 16th IEEE International Conference on Program Comprehension (ICPC'08)*, pages 123–132. IEEE, 2008.

[42] J. Stylos and B. A. Myers. Mica: A web-search tool for finding API components and examples. In *Proceedings of the Visual Languages and Human-Centric Computing (VLHCC'06)*, pages 195–202, 2006.

[43] I. Sutskever, O. Vinyals, and Q. V. Le. Sequence to sequence learning with neural networks. In *Advances in neural information processing systems (NIPS'14)*, pages 3104–3112, 2014.

[44] 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 (FSE'14)*, pages 269–280. ACM, 2014.

[45] L. Van Der Maaten. Accelerating t-sne using tree-based algorithms. *The Journal of Machine Learning Research*, 15(1):3221–3245, 2014.

[46] J. Wang, Y. Dang, H. Zhang, K. Chen, T. Xie, and D. Zhang. Mining succinct and high-coverage API usage patterns from source code. In *Proceedings of the 10th Working Conference on Mining Software Repositories (MSR'13)*, pages 319–328. IEEE Press, 2013.

[47] S. Wang, T. Liu, and L. Tan. Automatically learning semantic features for defect prediction. In *Proceedings of the 38th International Conference on Software Engineering (ICSE'16)*, pages 297–308. ACM, 2016.

[48] M. White, C. Vendome, M. Linares-Vásquez, and D. Poshyvanyk. Toward deep learning software repositories. In *Proceedings of the IEEE/ACM 12th Working Conference on Mining Software Repositories (MSR'15)*, pages 334–345. IEEE, 2015.

[49] T. Xie and J. Pei. MAPO: Mining API usages from open source repositories. In *Proceedings of the 2006 international workshop on Mining software repositories (MSR'06)*, pages 54–57. ACM, 2006.

[50] J. Yang and L. Tan. Inferring semantically related words from software context. In *Proceedings of the 9th IEEE Working Conference on Mining Software Repositories (MSR'12)*, pages 161–170. IEEE Press, 2012.

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