Mining Likely Analogical APIs across Third-Party Libraries via Large-Scale Unsupervised API Semantics Embedding

Chunyang Chen, Zhenchang Xing, Yang Liu, Kent Ong Long Xiong.

Abstract—Establishing API mappings between third-party libraries is a prerequisite step for library migration tasks. Manually establishing API mappings is tedious due to the large number of APIs to be examined. Having an automatic technique to create a database of likely API mappings can significantly ease the task. Unfortunately, existing techniques either adopt supervised learning mechanism that requires already-ported or functionality similar applications across major programming languages or platforms, which are difficult to come by for an arbitrary pair of third-party libraries, or cannot deal with lexical gap in the API descriptions of different libraries. To overcome these limitations, we present an unsupervised deep learning based approach to embed both API usage semantics and API description (name and document) semantics into vector space for inferring likely analogical API mappings between libraries. Based on deep learning models trained using tens of millions of API call sequences, method names and comments of 2.8 millions of methods from 135,127 GitHub projects, our approach significantly outperforms other deep learning or traditional information retrieval (IR) methods for inferring likely analogical APIs. We implement a proof-of-concept website (https://similarapi.appspot.com) which can recommend analogical APIs for 583,501 APIs of 111 pairs of analogical Java libraries with diverse functionalities. This scale of third-party analogical-API database has never been achieved before.

Index Terms—Analogical API, Word embedding, Skip thoughts

1 Introduction

Third-party libraries are an integral part of many software systems [1]. Due to various software maintenance reasons, developers of a software project often need to migrate the project from a currently-used library to some other libraries [2]. This migration activity is referred to as *library migration*. To complete a library migration task, the developer needs to first find a good analogical library that provides comparable features as the currently-used library. In this setting, we refer to the currently-used library as *source library*, and the analogical library as *target library*. Then, developers must establish the mappings of analogical APIs (methods in this work) between the libraries that implement the same or similar feature. Although analogical libraries can be effectively recommended [3], establishing analogical API mappings between libraries lacks effective support. Without the knowledge of analogical API mappings, it is impossible to migrate the code from using the source library APIs to using the target library APIs.

Some API mapping databases have been created by domain experts well versed in the source and target APIs, but only for a few source/target platform pairs, such as Android Qt to Windows 7 [4]. For an arbitrary pair of source and target libraries, developers have to manually find analogical APIs between the two libraries by studying and comparing API semantics. According to our observation, developers will first look for analogical APIs with similar names, and then they check the documentation to see if the candidate APIs have desired functionality. If they are still not sure, they may further check the API usage pattern to determine whether the candidate API is the analogical one. However, this manual investigation process is tedious and error-prone [5], especially when developers are unfamiliar with the target library and the target library has a large number of APIs [6], [7].

Having an automatic technique to create a database of likely API mappings between analogical source-target libraries can significantly ease the task. However, existing techniques have two fundamental limitations. First, existing techniques [8], [9], [10] require already ported or functionality similar applications for inferring likely API mappings between libraries. As a library often has several analogical libraries [3], it is unlikely to have already ported or functionality similar applications for an arbitrary pair of analogical source-target libraries. Second, some techniques [11] measure the textual similarity of the source and target API names/documents to discover likely API mappings. However, the descriptions of analogical APIs written by developers of different libraries may share few words in common (i.e., have lexical gap). For example, the two analogical APIs of *slf4j* and *log4j* have very different descriptions, *"Clear all entries in the MDC of the underlying implementation"* versus *"Remove all values from the MDC"*. Traditional IR methods are not robust to recognize API mappings when API names/documents have lexical gaps (more examples in Table 1).

• *Chunyang Chen is with Faculty of Information Technology, Monash University, Australia. Zhenchang Xing is with College of Engineering & Computer Science, Australian National University, Australia. Yang Liu and Kent Ong Long Xiong are with SCSE, Nanayng Technological University, Singapore. E-mail: chunyang.chen@monash.edu, zhenchang.xing@anu.edu.au, yangliu@ntu.edu.sg, kent0002@e.ntu.edu.sg*

Manuscript received December 19, 2017;.

To overcome the above limitations in creating the database of likely API mappings for analogical libraries, we present an approach that exploits unsupervised deep learning techniques to embed API usage semantics and API description semantics to infer likely API mappings between a pair of analogical libraries. In this work, we focus on API mappings between a source API method and a target API method of Java libraries. Our key technical contributions are two-fold:

Embedding API usage semantics by unsupervised API usage embeddings. Although already ported or functionality similar applications for a pair of analogical libraries rarely exist, there are usually many projects using one of the libraries. Our key observation is that these projects are not functionally similar as a whole, but analogical APIs are often used in a smaller similar context (e.g., method-level API call sequence) in different projects. For example, the two analogical APIs FileUtils.forceMkdir() in *Apache Commons IO* and Files.createParentDirs() in *Guava IO* are often used with Java file IO APIs like java.io.File.exists(), java.io.File.getParentFile(), or third-party APIs like LoggerFactory.logger.error() of *SLF4J*. This observation inspires us to adopt word embedding techniques [12], [13] to learn API usage embeddings using an API's surrounding APIs in API call sequences in an unsupervised way.

Embedding API description semantics by unsupervised skip-thoughts sentence vectors. To bridge the lexical gap in measuring API description similarity, we leverage Recurrent Neural Network (RNN) which is able to quantify text similarity in the presence of lexical gap. As we do not have priori knowledge of analogical APIs to compile pairs of API descriptions for supervised training of a RNN model, we adopt an unsupervised RNN model (skip thoughts [14]) which encode sentence semantics by not only words in the sentence but also the surrounding sentences around it (i.e., context) in documents. The trained skip thoughts models are used to embed API names and documents to sentence vectors respectively.

Our approach combines the three aspects of API similarities to infer likely API mappings between analogical source-target libraries, i.e., the similarity of API name vectors, the similarity of API document vectors, and the similarity of API usage embeddings. For each API method in the source library, our approach outputs a ranked list of API methods in the target library that are likely analogical APIs to the source API.

As a proof of concept, we choose 97 Java libraries and their analogical Java libraries (in total 111 pairs of analogical source-target libraries and 583,501 API methods) from the knowledge base of analogical libraries [3], [15]. These libraries support a diverse set of functionalities, including testing, data visualization, GUI development, information retrieval, etc. We discover likely analogical APIs between these pairs of analogical libraries and build a web application for searching analogical APIs across these pairs of libraries.

Using the 164 pairs of ground-truth analogical APIs manually identified in a previous work [16], we show that our approach outperforms baselines using other deep learning methods (e.g., topical word embedding [17], mean word embedding [18]) or traditional IR metrics (Term Frequency/Inverse Document Frequency, TF-IDF). Our manual examination of the recommended analogical APIs for 384 randomly selected APIs in 12 pairs of analogical libraries demonstrates the generality of our approach for libraries with diverse functionalities.

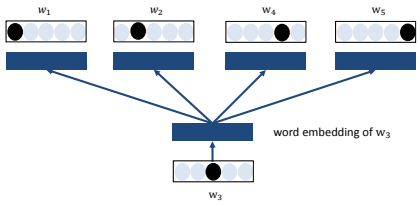


Fig. 1. Continuous skip-gram model. It predicts surrounding words given the center word.

Our contributions are listed as follows:

- We present the first unsupervised approach to incorporate both API usage and description similarities to create the database of likely analogical API mappings of third-party libraries.
- We conduct both quantitative and manual evaluation to evaluate the effectiveness of our approach.
- We build a large database of analogical API mappings for 111 pairs of analogical libraries and implement a web application[1] for accessing the database.

2 Background

We first introduce the two key techniques, i.e., word embeddings and recurrent neural network that our approach relies on.

2.1 Word Embeddings

Word embeddings are dense low-dimensional vector representations of words that are built on the assumption that words with similar meanings tend to be present in similar context. Studies [19], [20], [21] show that word embeddings are able to capture rich semantic and syntactic properties of software-specific words for measuring word similarity.

Continuous skip-gram model [12] is an efficient algorithm for learning word embeddings using a neural network model. Fig. 1 shows the network structure to train a continuous skip-gram model. Given a corpus of training word sequences, the goal of the model is to learn the word embeddings of a center word (i.e., w_i) that is good at predicting the surrounding words in a context window of $2t+1$ words ($t = 2$ in this example). The objective function is to maximize the sum of log probabilities of the surrounding context words conditioned on the center word:

$$\sum_{i=1}^{n} \sum_{-t \le j \le t, j \neq 0} \log p\left(w_{i+j} | w_i\right) \quad (1)$$

Intuitively, $p\left(w_{i+j} | w_i\right)$ estimates the normalized probability of a word w_{i+j} appearing in the context of a center word w_i over all words in the vocabulary. This probability can be efficiently estimated by the negative sampling method [13].

Continuous skip-gram model maps words onto a low-dimensional, real-valued vector space. Word vectors are

1. https://similarapi.appspot.com

essentially feature extractors that encode semantic and syntactic properties of words in their dimensions. In this vector space, semantically-similar words (e.g., $woman$ and $lady$) are likely close in term of their vectors' cosine similarity. Furthermore, pairs of analogical words (e.g., "man:woman" and "king:queen") are likely similar in terms of relations between pairs of words which can be computed by vector offsets between word pairs (e.g., $man - woman \approx king - queen$).

2.2 Recurrent Neural Network

Recurrent Neural Network (RNN) is a class of neural networks where connections between units form directed cycles. Compared with traditional n-gram language model [22], an RNN-based language model can predict a next word by preceding words with variant distance rather than a fixed number of words. Due to this nature, it is especially useful for tasks involving sequential inputs such as speech recognition [23] and code completion [24]. A basic RNN model includes three layers. An input layer maps each word to a vector using word embedding or one-hot word representation. A recurrent hidden layer recurrently computes and updates a hidden state after reading each word. An output layer estimates the probabilities of the next word given the current hidden state.

Complex RNN-based models have been developed for Natural Language Processing (NLP) tasks. For example, the RNN encoder-decoder model [25], [26], [27] is commonly adopted for machine translation tasks. This model includes two RNNs: one RNN to encode a variable-length sequence into a fixed-length vector representation, and the other RNN to decode the given fixed-length vector representation into a variable-length sequence. From a probabilistic perspective, this model is a general method to learn the conditional distribution over a variable-length sequence conditioned on yet another variable-length sequence, i.e., $P(y_1, ..., y_{T'}|x_1, ..., x_T)$. The length of the input T and output T' may differ.

The architecture of the RNN encoder-decoder model can be seen in Fig. 2 (The example is for English-to-French machine translation). The encoder is an RNN that reads each word of an input English sentence x sequentially. As it reads each word, the hidden state of the RNN encoder is updated. After reading the end of the the input (marked by an end-of-sequence symbol), the hidden state of the RNN is a vector c summarizing the whole input English sentence. The decoder of the model is another RNN which is trained to generate the output French sentence by predicting the next word y_t given the hidden state $h_{(t)}$ and the summary vector c. The two RNN components of the RNN encoder-decoder model are jointly trained to maximize the conditional log-likelihood

$$\max_{\theta} \frac{1}{N} \sum_{n=1}^{N} \log p_\theta(y_n|x_n) \quad (2)$$

where θ is the set of the model parameters and each (x_n, y_n) is a pair of input and output sequence from the training corpus.

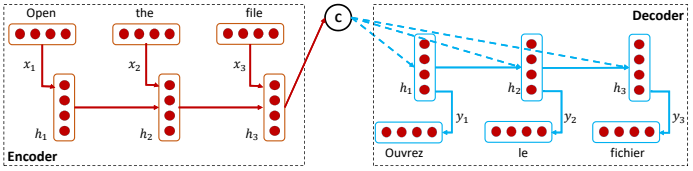


Fig. 2. RNN encoder-decoder model

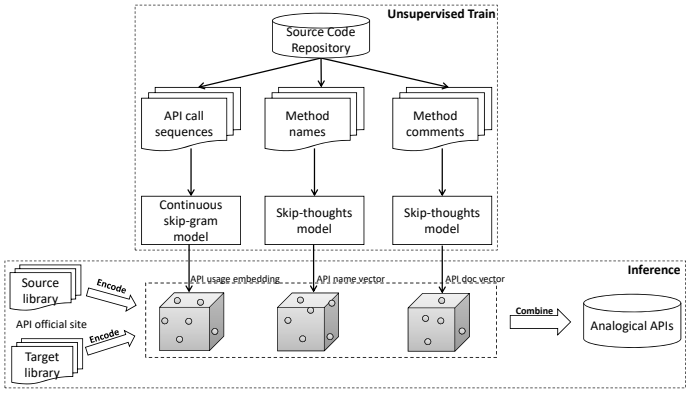


Fig. 3. The overall framework of our approach

3 The Approach

Fig. 3 depicts the overall framework of our approach. Our approach consists of an unsupervised training phase and an inference phase. In the training phase, we crawl a source code repository and extract API call sequences, method names and comments from source code. We consider APIs as words and apply continuous skip-gram model to a corpus of API call sequences to learn API usage embeddings. Furthermore, based on the corpus of method names (or comments), we train a name (or comment) skip-thoughts model [14]. Skip-thoughts model essentially incorporates the unsupervised learning mechanism of continuous skip-gram model [13] into the RNN encoder-decoder architecture [25].

In the inference phase, our approach creates a database of likely API-method mappings between a source library (L_s) and an analogical target library (L_t). Each likely mapping involves a source API method of L_s and a target API method of L_t. Our approach uses the learned API embeddings, the name skip-thoughts model and the comment skip-thoughts model to obtain API usage embeddings, API name vectors and API comments vectors. The likelihood of analogical API mappings is then determined by combining API usage, name and document similarities.

3.1 Encoding API Usage Similarity

We develop a word-embedding based method to embed API usage semantics in a vector space based on the use of an API, together with other APIs (not necessarily from the same library), in API call sequences extracted from the code. The obtained API vectors are then used to quantify API usage similarity between a source API method and a target API method.

3.1.1 *Extracting API Call Sequences in the Code*

To apply word embedding techniques to embed API usage semantics to vector, a large corpus of API call sequences has

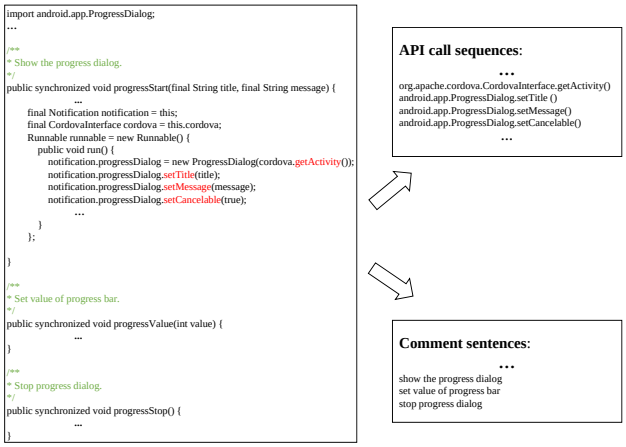


Fig. 4. An example of extracting API call sequence from a method and extracting comment sentences from one source code file.

to be prepared. We crawl a large corpus of source code from GitHub. Intra-method API call sequences are then extracted from the crawled source code, i.e., one API call sequence per method declaration, as illustrated in Fig. 4. Although technical details vary for extracting API call sequences from code written in different programming languages, we follow two principles in data crawling and preparation.

• **Extracting all API usage data:** If we crawl only projects that use the source (or target) library and extract API call sequences that contain only the APIs of the source (or target) library, the learned API vectors capture the semantics of an API only relative to other APIs in the same library, and thus are independent of the API vectors of the other library. Such API vectors cannot be used to infer likely API mappings between libraries. Therefore, the extracted API call sequences should include all APIs used in a method, no matter they are from the source (or target) library or any other libraries (as the example shown in Fig. 4).

When APIs of other libraries are used together with the APIs of the source (or target) library, they essentially provide a direct common context to bridge the otherwise independent source and target APIs. Even when APIs of other libraries are not directly used with source (or target) APIs, they could still provide indirect common context to learn API usage semantics through the propagation mechanism of the underlying neural network. To better capture semantics of APIs of other libraries which will directly or indirectly contribute to the embeddings of source (or target) APIs, projects that do not use the source (or target) library but use other libraries should also be crawled and analyzed.

• **Normalizing API mentions**: Our approach adopts word embedding techniques developed for natural language sentences to API call sequences. NLP tasks assume that the same words are used when a particular concept is mentioned in sentences. When applying word embedding techniques to API call sequences, we need to make sure that this assumption holds.

First, a simple name may refer to several different APIs. In NLP, this simple name has sense ambiguity. For example, getValue() is a widely used method name even for different classes in one library. If we cannot distinguish them from one another, we cannot accurately learn their semantics. To disambiguate methods with such simple names, fully-qualified names should be used in the extracted API call sequences. Extracting fully-qualified API names in the code usually requires the code to be compilable. However, it requires prohibitive efforts to make hundreds of thousands of projects crawled from GitHub compilable, for example to fix all missing library dependencies. Therefore, we adopt the partial program analysis techniques tool[2] [28] to extract fully-qualified API names from non-compilable code.

Second, method overloading allows two or more APIs to have the same name but different parameter lists. As overloading methods have the same name within the same class, they provide the same functionality at the conceptual level [29], [30], [31]. To validate this assumption, we compare the method descriptions of the overloading methods in our dataset. Among the 583,501 APIs from 111 pairs of analogical Java libraries in our study, we identify 31,095 groups of overloading APIs with the same method descriptions, and 7,694 groups of overloading APIs with different descriptions. This analysis shows that over 80% of the overloading methods have the same document description, i.e., the same conceptual-level functionality. Furthermore, the percentage of the overloading APIs with the same descriptions is highly underestimated because some different descriptions of the overloading APIs actually convey the same meaning, e.g., the overloading methods of "org.geotools.geometry.jts.WKTWriter2.toLineString()" contain two different but same-meaning descriptions "Generates the WKT for a LINESTRING specified by two Coordinates", and "Generates the WKT for a LINESTRING specified by a CoordinateSequence". Therefore, we normalize the call of overloading APIs as one API by ignoring their parameter lists. This also helps to mitigate the data sparsity issue when overloading APIs are treated as different APIs.

3.1.2 Learning API and Library Embeddings

We consider an API call sequence as a sentence, and each API method as a word. APIs with different fully-qualified names are treated as different words in sentences, and overloading APIs are treated as the same word. We use continuous skip-gram model [13] to learn the vector representation of each API (i.e., API embedding) based on the surrounding APIs of an API (i.e., the APIs called before and after the API) in the corpus of API call sequences. As API call sequences are usually short (5.3 on average, 4 on median in our dataset), we set the size of context window at 5 (i.e., $t = 2$ in Eq. 1). Given a library, we find all APIs of this library by crawling the library's official API website (more details in Section 3.2.3). We take the average of the API vectors of all APIs of this library as the library embedding.

Relational similarity between pairs of words is a metric for reasoning about analogical words [12], [13]. Chen et al [3] shows that library-language relational similarity (e.g., "NLTK:Pyton" and "OpenNLP:Java") performs better in inferring analogical libraries than directly using the cosine similarity of libraries, based on the tag embeddings learned from short and diverse question tag sequences.

3.1.3 Measuring API Usage Similarity

We find that this observation also extends to analogical APIs between libraries. For example, Fig. 5 illustrates two

2. http://www.sable.mcgill.ca/ppa/

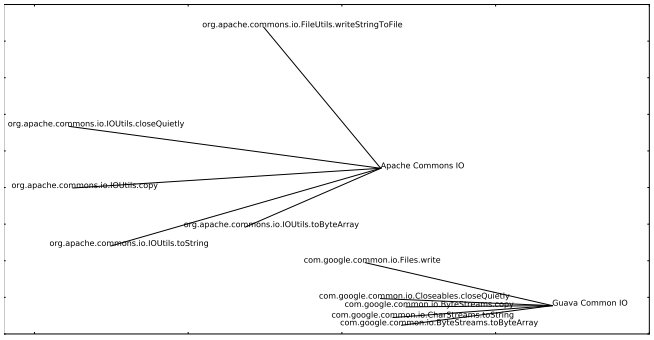


Fig. 5. Two-dimensional PCA projection of API and library embeddings. Parallel lines reveal analogical API-library relations.

analogical libraries (*Apache Commons IO* and *Guava Common IO*) and five APIs of each library. The API and library embeddings are projected into a two-dimensional vector space using Principle Component Analysis [32], a technique commonly used to visualize high-dimensional vectors. Note that a library embedding is computed by averaging the embeddings of all this library's APIs, not just the five APIs shown in the figure. In Fig. 5, relation between an API and its library is visualized as a line linking the API and the library, i.e., the position of an API relative to the position of its library in the vector space. We can see that analogical APIs exhibit roughly parallel lines in the API-library relations. The more parallel, the more similar the usage of analogical APIs relative to their libraries.

Therefore, in our approach, we use API-library relational similarity to measure the usage similarity between an API (a_s) in the source library (L_s) and an API (a_t) in the target library (L_t). This relational similarity can be computed by vector arithmetic as follows:

$$sim_{use} = \cos(uvec(a_s) - uvec(L_s), uvec(a_t) - uvec(L_t)) \tag{3}$$

where $uvec()$ (usage vector) is API embedding or library embedding, vector offset of API embedding and library embedding reflects the API-library relation (line) in the vector space, and relational similarity is computed as cosine similarity between the two vector offsets (i.e., the angle between the two API-library lines).

3.2 Encoding API Description Similarity

Word embedding models embed only word-level semantics. To represent an API name (or document) (i.e., a sentence of multiple words) as a vector, we adopt skip thoughts [14] (an unsupervised RNN model) to encode API names (or documents) into a vector space. Due to the power of the RNN model, the obtained API name (or document) vectors can quantify the semantic similarity in the name (or document) of the two APIs, even in the presence of lexical gap. We exploit documentation guidelines and naturalness of source code [33] to prepare a large corpus of code-related text to train the skip thoughts model.

3.2.1 Skip Thoughts Model

Skip-thoughts model [14] is a recently proposed unsupervised RNN model that encodes the semantics of a sentence to vector by not only the words in a sentence, but also

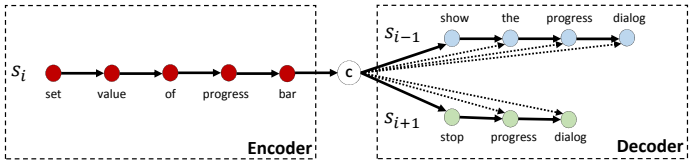


Fig. 6. Skip-thoughts model. It predicts surrounding sentences given the center sentence.

the surrounding sentences of the sentence in a document. Fig. 6 illustrates the architecture of the skip-thoughts model. Skip-thoughts model consists of an RNN encoder and two RNN decoders. The encoder encodes words of a sentence to a sentence vector. The two decoders decode this sentence vector to the previous sentence and the next sentence in a document, respectively. Skip-thoughts model is inspired by the continuous skip-gram model [13]. The difference is, instead of using a word to predict its surrounding words, skip-thoughts model encodes a sentence to predict the surrounding sentences.

Given a sentence tuple (s^{i-1}, s^i, s^{i+1}) in a document, let w_t^i denote the t-th word for sentence s^i and let x_t^i denote its word embedding. The encoder in the model outputs a sentence vector $\mathbf{c}$ from the sentence s^i. One decoder is used to generate the next sentence s^{i+1} from the sentence vector $\mathbf{c}$, while the other decoder is used to generate the previous sentence s^{i-1}. The different colors in Fig. 6 indicate which components share the same parameters. The objective function is to minimize the sum of the log-probabilities for the previous sentence s^{i+1} and the next sentence s^{i-1} conditioned on the encoder's output sentence vector $\mathbf{c}$ over all training tuples:

$$\begin{aligned} &\sum_t \log P(w_t^{i+1} | (w_1^{i+1}, ..., w_{t-1}^{i+1}), \mathbf{c}) \\ &+ \sum_t \log P(w_t^{i-1} | (w_1^{i-1}, ..., w_{t-1}^{i-1}), \mathbf{c}) \end{aligned} \tag{4}$$

The skip-thoughts model tries to reconstruct the surrounding sentences of an encoded sentence. Thus, sentences that share similar surrounding contexts are mapped to similar vector representations. This helps to mitigate the lexical gap issue in quantifying sentence similarity. Some examples are shown in Table 1, in which we take a method declaration comment as query to find the most similar method declaration comment in the vector space embedded using skip-thoughts model. We can see that semantically-similar comments can be retrieved even if they have lexical gaps. Traditional IR methods (e.g., TF-IDF) cannot reliably quantify text similarity in the presence of such lexical gaps.

3.2.2 Extracting Code-Related Text

To train a skip-thoughts model to embed API-method names (or documents) to vector, a large corpus of code-related text is required. As the name (or document) of an API describes the functionality of the API, the training code-related text should contain sentences serving the similar purpose. Therefore, we extract a method-name corpus and a method-comment corpus from the source code crawled from GitHub, which is used to train a method-name skip-thought model and a method-comment skip-thought model, respectively.

TABLE 1
Examples of lexically different but semantically similar method declaration comments from GitHub code

Query	Nearest sentence
concatenate all of the strings	joins the array of strings
checks if a file exists	tells whether the file exists
performs the dijkstra algorithm	this is used in the path finding algorithm
reads all the exif data from the input file	parses the exif data from the specified file
remove the action icon from the given index	remove the action at the specified index
closes an open db connection	closes the current database connection
prints out all valid moves at the given position	prints all legal moves in a position
returns the number of items in the map	returns the size of the map

According to the learning mechanism of skip-thoughts model, the training corpus should contain documents that contain a sequence of sentences. In this work, we construct a method-name (or method-comment) document for each code file, because a code file should group a set of closely-related methods. Method declaration comments may contain several sentences. According to documentation guidelines (e.g., Javadoc guidance [34]), the first sentence in method declaration comments should provide a concise description of a method's functionality. Also considering the high training complexity of skip-thoughts model, we extract only the first sentence in the method declaration comments. A method declaration without comment will be ignored. If the first sentence matches one of the six patterns, i.e., beginning with "TODO:", "Created by IntelliJ IDEA", "Created with IntelliJ IDEA, "User:", "Data:", "Name:", we consider the comment as an irregular comment and ignore the corresponding method.

Inspired by the studies on the naturalness of source code [33], [35], [36], [37], we order the names (or comment sentences) extracted from methods declared in a code file in the same order as method declarations. The intuition is that developers declare methods in an order according to certain natural relatedness among methods, and this natural relatedness is reflected in the order of the names (or comments) describing the methods' functionalities. For example, the source code file in Fig. 4 is an implementation of an Android application. We can see that the name (or comment) of "progressValue" method is highly related to the name (or comment) of its preceding method "progressStart" and the name (or comment) of its following method "progressStop" as this is the logic flow to implement a progress bar. That is, the surrounding names (or comments) of a name (or comment) provide the context to embed the semantics of the given name (or comment).

3.2.3 Measuring API Name (or Document) Similarity

Given a library, we manually locate its official API website and then use a web crawler to crawl all API web pages from the website. We extract the fully-qualified API name and the first sentence of the API description for each API from its webpage. Then, we use the trained method-name (or method-comment) skip-thoughts model to embed API names (or documents) in a vector space.

For API name, we split the name into a sequence of words according to API naming convention. For example, for an API method com.googlecode.charts4j.Plots.newRadarPlot() from the library *charts4j*, we split the name by "." and camel case, and remove common package prefix "com", "googlecode", "charts4j" which appear in all API names[3], and normalize the words to lower case. We obtain a sentence "plots new radar plot" for the API, which is entered to the skip-thoughts model to obtain the name vector for the API. Given the name vector $nvec$ of the two APIs, a_s in the source library L_s and a_t in the target library L_t, we use cosine similarity to measure the similarity of the two API names:

$$sim_{name}(a_s, a_t) = cos(nvec(a_s), nvec(a_t)) \quad (5)$$

For the first sentence of the API description, we directly enter it to the trained method-comment skip-thoughts model to obtain the document vector for the API. Given the document vector $dvec$ of the two APIs, a_s in the source library L_s and a_t in the target library L_t, we use cosine similarity to measure the similarity of the two API documents:

$$sim_{doc}(a_s, a_t) = cos(dvec(a_s), dvec(a_t)) \quad (6)$$

3.3 Building Analogical-API Knowledge Base

Given a source library L_s, we search the knowledge base of analogical libraries [3] to find its analogical libraries (i.e., the target library L_t). We determine the likelihood of an API method a_s in the source library L_s and an API method a_t in the target library L_t being analogical by combining three information similarities of the two APIs, i.e., the similarity of API names (sim_{name}), the similarity of API documents (sim_{doc}), and the similarity of API usage (sim_{usage}) into an overall similarity score:

$$Sim(a_s, a_t) = \alpha \times sim_{usage}(a_s, a_t) + \beta \times sim_{doc}(a_s, a_t) + \gamma \times sim_{name}(a_s, a_t) \quad (7)$$

where α, β, γ are the weight parameters in $(0, 1)$, and we make $\alpha + \beta + \gamma = 1$ so that the final similarity value is of the range $[-1, 1]$ where 1 is the most similar and -1 is the opposite. These weight parameters are tuned using a small evaluation set of analogical API mappings in the tool implementation. We will elaborate the parameter tuning in Section 5.2.3.

For each API method in the source library, we rank the API methods in the target library by their overall similarity with the given source API. We take the top 10 target API methods with the highest similarity values as the likely analogical APIs to the source API method. Then, we obtain a knowledge base of likely analogical API mappings for a pair of analogical source-target libraries.

For skip-thought model, we only take words with frequency greater than 5 into consideration. In this setting, the vocabulary size is 13,067 for API description, and 9,569 for API names. Considering the data nature and training efficiency, we set the number of context sentences as 1. That is when training the skip-thought model, the current sentence will infer 1 prior sentence and 1 subsequent sentence. The encoder and decoder in our skip-thought model contain 1-layer RNN respectively.

4 Tool Support

We have implemented our approach in a proof-of-concept analogical-APIs search web application (https://similarapi.appspot.com). This web application contains the database

3. Common package prefixes for all libraries in this study are listed in https://similarapi.appspot.com/packagePrefix.html

of likely analogical APIs between 111 pairs[4] of analogical Java libraries. The database includes the APIs of the latest version of these Java libraries. To train our skip-thought model and API embedding model, we downloaded 135,127 Java projects from Boa dataset [38], [39]. Boa (http://boa.cs.iastate.edu/) is a domain-specific language and infrastructure that eases mining software repositories. The Boa dataset has also been used in several studies on mining software repositories [40], [41], [42] In our work, we remove few-star (< 10) or deleted projects and obtain 135,127 Java projects as our dataset. These projects contain 2,058,240 source-code files. We extract API call sequences from the crawled code using a partial program analysis (PPA) [28] tool for Java. We collect a corpus of 10,554,900 API call sequences which includes 952,829 unique APIs for learning API embeddings. We use the Eclipse JDT compiler [43] to extract method names and comments (Javadoc format) from the crawled code. We obtain a corpus of 26,622,034 method names and a corpus of 2,798,837 comments for training the method-name and method-comment skip-thoughts models respectively. Note that method declarations always have a name, but they may not have comment and we also ignore those with irregular comments. The parameter tuning for the tool will be discussed in Section 5.2. The trained skip-thought model and API embedding model are adopted to infer the API usage and description similarity of analogical APIs among 583,501 APIs of 111 pairs of analogical Java libraries.

5 Evaluation

This section reports the quantitative and qualitative evaluation of our approach, which aims to answer three research questions:

- RQ1: How do different parameter settings affect the performance of our approach?
- RQ2: How well can API usage and description similarities determine analogical APIs independently or as a whole?
- RQ3: How well can our approach work for libraries with diverse functionalities?

5.1 Experiment Setup

First, we describe how we collect the ground truth for the evaluation and the metrics used in the evaluation.

5.1.1 Ground Truth of Analogical-API Mappings

To evaluate our approach automatically, we collect a set of ground-truth analogical API mappings from the dataset of API mappings[5] released by Teyton et al. [16]. This dataset involves 4 pairs of similar libraries: (Apache Commons IO [44], Guava IO [45]), (Apache Commons Lang [46], Guava Base [47]), (JSON [48], gson [49]) and (mockito [50], jMock [51]). For each pair of libraries, Teyton et al. manually validate the discovered API mappings and produce a set of ground-truth API mappings. Teyton et al.'s work considers overloading APIs as different APIs, while our work considers overloading APIs as one API. Therefore, we merge the mappings involving overloading APIs in Teyton's dataset as one mapping in our evaluation. We obtain in total 82 API mappings (see Table 2) as ground truth to evaluate our approach. Each mapping has a pair of API methods, one from each analogical library. As API mappings are mutual, we can use either one API method in a mapping as query API and the other one as analogical API to be recommended. Therefore, we have $82 \times 2 = 164$ query APIs for the evaluation.

TABLE 2
Ground Truth Summary

Analogical library pair	#API mappings	Functionality
Apache Commons IO/Guava IO	14	I/O
Apache Commons Lang/Guava Base	30	Utility methods
JSON/gson	22	(De)serialization
mockito/jMock	16	Test mocking

5.1.2 Evaluation Metrics

In our experiment, we adopt two metrics to measure the performance of our analogical-APIs recommendation results: Recall rate@k (Re@k) and Mean Reciprocal Rank (MRR). The recall rate@k (Re@k) is commonly used to evaluate recommendation systems used in software engineering [1], [52], [53]. For each query API a_i, let the ground truth analogical API be gt_i, and let the set of top-k recommended APIs be R_i. The recall rate@k (Re@k) is the proportion of recommendation sets R_i for all query APIs that include the corresponding ground truth API gt_i. We use a small value for $k = 1, 5, 10$ as developers are unlikely to look through a long recommendation list.

In detail, the Re@1 means the probability of the first recommendation API from our model is just the ground truth. Similarly, Re@10 represents the probability that the ground truth can appear in our top-10 recommendation list. Note that we set the maximum k in this work at 10 as developers are unlikely to look through a long recommendation list. Compared with the recall rate in the top 10 recommendation results, developers sometimes may be more concerned with the position of first real analogical API within the recommendation list. Therefore, apart from Re@k, we also adopt MRR which only cares about the single highest-ranked relevant item as an evaluation metric in this work. MRR is a commonly adopted measure for evaluating the performance of information retrieval systems especially API recommendation [54], [55], [56]. Let k be the rank position of the ground-truth analogical API in the recommendation list for a query API, then the reciprocal rank (RR) is defined as $\frac{1}{k}$. The MRR is the mean of the RRs over all query APIs. The higher the MRR and Re@k metrics are, the better the recommendation results are.

5.2 RQ1: Impact of Parameter Settings

Next, we report the experiments of different parameter settings that affect the performance of our approach. Through these experiments, we determine the parameter settings used for our approach in the evaluation as well as for our tool implementation.

4. All pairs can be seen in https://similarapi.appspot.com/allLibPair.html

5. http://web.archive.org/web/20160412155655/, http://www.labri.fr/perso/cteyton/Matching/index.php

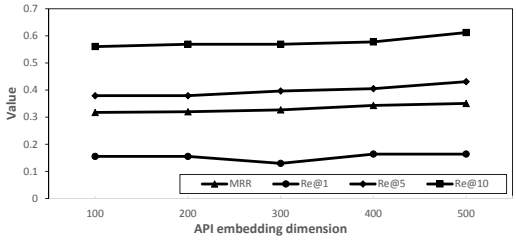


Fig. 7. The performance of encoding API usage similarity with different API embedding dimensions

5.2.1 API Embedding Dimensions

The most important parameter for the continuous skip-gram model [13] is the dimension of the API vectors to be learned. We test five different API embedding dimensions from 100 to 500 with the increment 100 which is commonly adopted in other works [57], [58] using word embedding in SE context. For each dimension setting, we use the learned API vectors to recommend likely analogical APIs for each query API in the ground-truth based on the API usage similarity in Eq. 3. Fig. 7 shows that there are no significant differences in the performance in different dimension settings. But overall, the MRR and Re@k metrics increases slightly as the API embedding dimension increases. Therefore, we set the API embedding dimension at 500.

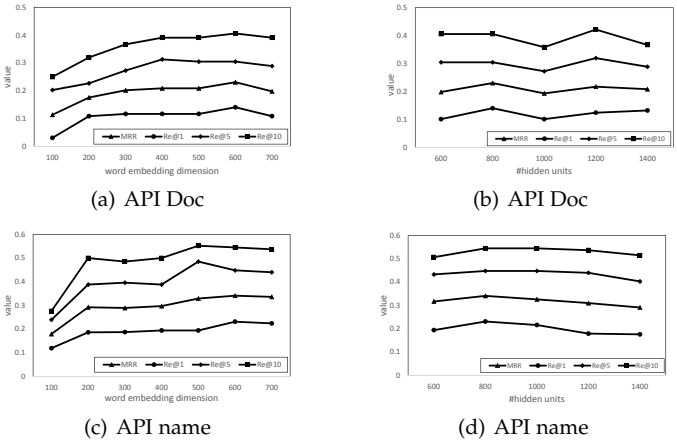
5.2.2 Skip-Thoughts Model Parameters

Skip-thoughts model has two key parameters, the dimension of input-layer word embedding and the number of the hidden units (sentence vector dimension) in the RNN model. To see the impacts of the parameters on encoding API name/document similarity, we vary the value of these two parameters with word embedding dimension from 100 to 700 while number of hidden units from 600 to 1400 which is commonly adopted in other software-engineering related works [58], [59].

Fig. 8 shows the performance differences in different parameter settings. We can see that when the dimension of the input-layer word embedding is 600, the skip-thoughts model achieves the best results for encoding both API name similarity (Fig. 8(c)) and API doc similarity (Fig. 8(a)). For the number of hidden units, the performance of 800 units is similar to that of 1200 units for encoding API doc similarity (Fig. 8(b)). But 800-units setting has better MRR (0.231 vs 0.218) and Re@1 (0.141 vs 0.125), compared with 1200-units setting. Furthermore, smaller unit number indicates faster training and prediction. The RNN model with 800 hidden units also produces the best performance to encode API name similarity (Fig. 8(d)). Therefore, we set the word embedding dimension at 600 and the number of RNN hidden units at 800.

5.2.3 Combining API Usage/Name/Document Similarities

As described in section 3.3, there are three weight parameters to combine the three similarities (API name similarity, API document similarity, and API usage similarity) into an overall similarity score. We perform the five-fold cross validation to select the value for these three weight parameters. In this experiment, we set API embedding dimension at 500, and word embedding dimension at 600 and the number of RNN hidden units at 800 for skip-thoughts model. We



(a) API Doc (b) API Doc

(c) API name (d) API name

Fig. 8. The performance of encoding API name/document similarity under different parameter settings

randomly divide the ground-truth API mappings into five equal-size portions for five-fold cross validation. For each iteration, we use four portions to determine the best combination of the three parameters and the rest one portion for testing.

We alter three parameters by 0.05 increments. For each combination of the three weight parameters, we recommend likely analogical APIs for each query API in the testing portion based on the overall similarity in Eq. 7. In our experiment, we observe that different library pairs may have different optimal weight parameters. However, considering the limited number of ground-truth analogical APIs and the generality of the weight parameters, we compute the MRR and Re@k metrics for the four pairs of ground-truth libraries as a whole. We obtain the best results in all MRR and Re@k metrics for the four pairs of ground-truth libraries, when we set α as 0.3 (for API usage similarity), β as 0.2 (for API document similarity), and γ as 0.5 (for API name similarity).

5.3 RQ2: Performance of API Usage and Description Similarties

In this section, we report the comparative study of our approach for analogical API recommendation and other deep learning methods or traditional information retrieval (IR) methods. We also study the impact of different combinations of API usage, name and/or document similarities on the recommendation performance.

5.3.1 The Performance of Encoding API Usage Similarity

First, we consider only API embeddings and recommend likely analogical APIs for a query API in the ground truth based solely on API usage similarity. Our method uses API-library relational similarity (see Eq. 3) to determine analogical APIs. We adopt two baselines to study the importance of library embeddings and different ways to incorporate API and library embeddings, respectively.

Baseline Methods:

- The first baseline is attributional similarity of API embeddings which is implemented in Nguyen et al.'s work [60]. In this baseline, we still use continuous skip-gram model to embed API usage in API call sequences to vector (see Section 3.1.2). But we recommend likely

TABLE 3
Comparison of analogical-API-recommendation performance by different methods to encode API usage similarity

Method	MRR	Re@1	Re@5	Re@10
Baseline1 (API attributional similarity)	0.294	0.164	0.405	0.56
Baseline2 (Topical API attributional similarity)	0.116	0.06	0.181	0.241
Our API-library relational similarity	**0.351**	**0.224**	**0.431**	**0.612**

analogical APIs for a query API based on the cosine similarity of the API embeddings between the query API and an API in the target library. In the first baseline, library embeddings is not taken into account at all when measuring API usage similarity.

- In the second baseline, we use a variant word embedding model, topical word embedding [17], to embed library semantics into API embeddings during the learning process. In our application of topical word embedding, we consider a library as the topic for its APIs. We prepare the corpus for training topical API embedding model by attaching the library to its APIs. After obtaining topical API embeddings, we use attributional similarity of topical API embeddings to recommend likely analogical APIs for a query API. The second baseline provides an alternative way to incorporate API and library embeddings, compared with our method.

Experiment Results: Table 3 shows the performance of our API usage similarity (API-library relational similarity) and the two baseline similarities (attributional similarity of API embeddings and attributional similarity of topical API embeddings). We can see that among the three similarity, our API usage similarity is significantly better than the other two baseline similarities for all evaluation metrics. Compared with attributional similarity of API embeddings, the MRR of our similarity is 0.35 (19.4% higher), recall rate@1 is 0.224 (36.6% higher) , recall rate@5 is 0.431 (6.4% higher). The reasons for the weaker performance of attributional similarity of API embeddings are twofold. First, API call sequences are shorter than natural language sentences. That is, there is often not much context to embed an API's usage. Second, the surrounding APIs of analogical APIs in API call sequences can be very diverse, depending on the features and implementation styles of the projects using analogical libraries. The context diversity may separate analogical APIs apart in the embedding space.

The performance of topical API embeddings is even much worse than that of the original API embeddings. This suggests that topical word embedding is not suitable for embedding library semantics with its APIs. In contrast, our API usage similarity is an effective mechanism to take into account library semantics in terms of an API's relation with its library in the vector space when reasoning about likely analogical APIs between libraries.

5.3.2 The Performance of Encoding API Name/Description Similarity

Next, we recommend likely analogical APIs for a query API in the ground truth using solely API name (or document) similarity. Our method uses the skip-thoughts model to obtain API name (or document) vector representations to determine their similarity. We compare our method with the two different ways to obtain API name (or document) vector representations.

Baseline methods: In this experiment, our method and the two baseline methods all measure API name (or document) similarity using the cosine similarity of API name (or document) vectors. To use the two baseline similarities, we build the corpus of all API Javadocs of the 111 libraries used in our current analogical-API search tool. Same as our approach, we collect only the first sentence of an API's Javadoc to build a corpus of API documents. We preprocess the fully-qualified API names in the same way as described in Section 3.2.3 and obtain a corpus of API names.

- The first baseline similarity is the traditional IR similarity based on the TF-IDF metric [61]. This baseline similarity is also used in Pandita et al.'s paper [11] to infer likely API mappings by the textual similarity of API documents. To use TF-IDF, we follow Pandita et al.'s work to preprocess the Javadoc corpus: stop-word removal, camel case split, lowercasing, stemming, etc. Given an API, we encode its API name (or Javadoc) to a TF-IDF vector.
- Apart from the traditional TF-IDF, we also set up some state-of-the-art sentence embedding algorithms based on the deep learning methods. The second baseline is based on the Doc2vec [62] model which composes word embeddings into sentence or document embeddings with neural networks. The third baseline is a sentence embedding model based on weighted average of word vectors in the sentence using PCA/SVD (Principal Component Analysis/Singular-Value Decomposition), and it is called smooth inverse frequency (SIF) [63]. Another baseline are Simple Word-Embedding based Models (SWEMs) [64] with three variations, consisting of parameter-free pooling operations such as average, max-pooling and concatenating the average and max-pooling embedding.

Experiment results: The comparison of different methods to encode Javadoc similarity is shown in Table 4. Doc2vec (Baseline 2) gets the poorest performance in both encoding API document similarity and name similarity. That may be because that Doc2vec is designed for document embedding, hence it does not work well for encoding sentences which are much shorter than document. Compared with SWEM (baseline 4, 5, 6), TF-IDF (baseline 1) and SIF (baseline 3) have much better performance in all metrics. But our skip-thoughts model outperforms all baselines in these two experiments. For encoding API document similarity, our skip-thoughts model gain 13.2%, 11.7%, 9.6% improvement in MRR, Re@5, Re@10 than the best results across all baseline, and it achieves the similar performance in Re@1 to SIF. For encoding API name similarity, our Skip-thoughts model gains 12.2%, 24.2%, 7.5% improvement in MRR, Re@1 and Re@10 than the best results across all baseline, and it achieves the similar performance in Re@5 to TF-IDF.

5.3.3 The Performance of the Combined Similarity

Finally, we compare the performance of recommending likely analogical APIs for a query API in the ground truth using a combination of API usage, name and/or document similarity. Table 6 summarizes the comparison results.

Although a sole similarity in our approach outperforms other deep learning or traditional IR baselines, any sole

TABLE 4
Comparison of analogical-API-recommendation performance by different methods to encode API document similarity

Method	MRR	Re@1	Re@5	Re@10
Baseline1 (TF-IDF)	0.202	0.148	0.273	0.344
Baseline2 (Doc2vec)	0.166	0.109	0.227	0.266
Baseline3 (SIF)	0.204	**0.156**	0.266	0.313
Baseline4 (SWEM-*avg*)	0.192	0.109	0.305	0.367
Baseline5 (SWEM-*max*)	0.155	0.109	0.211	0.305
Baseline6 (SWEM-*concat*)	0.157	0.117	0.211	0.25
Our skip-thoughts	**0.231**	0.141	**0.305**	**0.406**

TABLE 5
Comparison of analogical-API-recommendation performance by different methods to encode API name similarity

Method	MRR	Re@1	Re@5	Re@10
Baseline1 (TF-IDF)	0.303	0.186	0.448	0.507
Baseline2 (Doc2vec)	0.210	0.119	0.313	0.381
Baseline3 (SIF)	0.304	0.172	0.433	0.455
Baseline4 (SWEM-*avg*)	0.253	0.134	0.358	0.433
Baseline5 (SWEM-*max*)	0.257	0.134	0.373	0.440
Baseline6 (SWEM-*concat*)	0.257	0.134	0.381	0.455
Our skip-thoughts	**0.341**	**0.231**	**0.448**	**0.545**

similarity is not good enough for finding likely analogical API mappings. The MRR for using solely API usage, name or document similarity is about $0.23 \sim 0.35$, and the Re@1 is only $0.14 \sim 0.23$. That is, only for one or two out of 10 query APIs, the top-1 recommended API is the true analogical API. Combining any two similarities can boost the recommendation performance significantly with the MRR larger than 0.47 and the Re@1 larger than 0.31. Our overall similarity in Eq. 7 that incorporates all three similarities produces the best performance in all metrics. The MRR is 0.556 indicating that the analogical API can in average rank as the second position in the recommendation list. The Re@10 is 0.845, and the Re@1 is still reasonably high at 0.409. That is, for 4 out of 10 query APIs , the top-1 recommended API is indeed the true analogical API. For more than 8 out 10 query APIs, the true analogical APIs are included in the top-10 recommendation list.

These results indicate that API usage similarity and API description similarity can complement each other. Meanwhile, API name and document similarities are not completely redundant. That is, API names and documents can also provide complementary information to each other. Incorporating all three similarities can enhance the accuracy of analogical API recommendation task. Compared with the entire set of APIs in the target library, developers can highly likely find the analogical API by examining only a small number of APIs that our approach recommends.

We have compared our model with different baseline methods base on individual information i.e., encoding API usage (Section 5.3.1) and encoding API description and name (Section 5.3.2). According to the performance of the baselines, we then select the best baseline for encoding each kind of information (API attributional similarity for encoding API usage and SIF for encoding API description and name), and combine them to recommend analogical APIs. Although each component in our approach for encoding each kind of information has relatively small performance improvement than the best baseline, our overall model with the combined similarity achieves significant increase than the combined best baselines. As shown in Table 6, our model achieves 24.3%, 53.8%, 19.6%, 11.2% improvement in MRR, Re@1, Re@5, Re@10 compared with the combined best baselines.

TABLE 6
Comparison of analogical-API-recommendation performance by different combinations of API usage/name/document similarity

Information	MRR	Re@1	Re@5	Re@10
API usage	0.351	0.224	0.431	0.612
API document	0.231	0.141	0.305	0.406
API name	0.341	0.231	0.448	0.545
API usage+document	0.474	0.355	0.636	0.691
API usage+name	0.506	0.327	0.727	0.745
API document+name	0.470	0.318	0.636	0.827
API usage+document+name	**0.556**	**0.409**	**0.745**	**0.845**
API all info (baseline)	0.447	0.266	0.623	0.76

For API recommendation list, some candidates may be false positive results. Although developers need to filter out false positive recommendations from the top 10 results, this effort is much less than that to identify an analogical API out of hundreds or thousands of APIs of a library. In addition, considering the quality of our analogical API, if developers cannot find the actual analogical API in the top 10 recommended APIs, there would be very unlikely some analogical APIs for the query API. Therefore, even if no analogical API actually exists, using our approach can help developers quickly confirm the likelihood of non-existence of analogical APIs.

5.4 RQ3: Generality for Diverse Libraries

To validate the generality of our approach, we manually evaluate the performance of our analogical API recommendation for 384 randomly sampled query APIs in 12 pairs of analogical libraries.According to the widely-used sampling method [65], we examine the minimum number MIN of data instances in order to ensure that the estimated population is in a certain confidence interval at a certain confidence level. This MIN can be determined by the formula: $MIN = n_0/(1 + (n_0 - 1)/populationsize)$. n_0 depends on the selected confidence level and the desired error margin: $n_0 = (Z^2 * 0.25)/e^2$, where Z is a confidence level's z score and e is the error margin. For the final human evaluation, we examine MIN instances of relevant data for the error margin $e = 0.05$ at 95% confidence level i.e., $MIN = 384$. Therefore, we randomly sample 384 query APIs in 12 pairs of analogical libraries for manual evaluation.

5.4.1 Evaluation Procedure

In this evaluation, we use the four pairs of analogical libraries in the ground-truth plus the other 11 pairs of analogical libraries with diverse functionalities (see Table 7). Given a pair of analogical libraries, we consider one library as source library and the other as target library. To investigate our model's performance on unpopular libraries, we first define the popularity of the library. As all of the studied libraries appear in Stack Overflow tags which can be used to tag the topic of questions, we take the library tag frequency as the indicator of its popularity (before Aug 28, 2018). Among 96 libraries in this study, the median frequency is 695[6], so we regard libraries with higher frequency as "popular" while libraries with lower frequency as "unpopular". Then the 12 library pairs adopted in section 5.4 belong to

6. Detailed library frequency can be seen in https://similarapi.appspot.com/libFreq.html

three different groups i.e., 1) API mapping between popular libraries (4 pairs), 2) API mapping between hybrid pairs i.e., popular library and unpopular library (5 pairs), and 3) API mapping between unpopular libraries (3 pairs). We randomly sample 32 API methods in the source library as query APIs, and use our approach to recommend 10 API methods in the target library for each query API.

We recruit four PhD students in our school who have 3+ years Java programming experiences. Based on the participants' experience in different libraries, we assign each participant six pairs of analogical libraries so that each result is checked by two students. We calculate the Cohen's Kappa [66] to evaluate the inter-rater agreement. Examining one pair of analogical libraries takes a 40-minutes session with 10-minutes break between sessions. Participants are asked to determine whether a recommended target API is a true analogical API to the query API. Participants make their decision based on their priori knowledge of using the assigned libraries, API documents (if any), and/or other information (e.g., API usage examples) they can find on the Internet. Based on the participants' labeled true analogical APIs, we compute the MRR and Re@k for the recommendation results.

5.4.2 Mining Analogical APIs not in Ground-Truth API Mappings

The API mappings in the ground-truth contain only a small portion of APIs in the four pairs of analogical libraries, because Teyton et al's method [2] considers only APIs involved in code changes to port an application. For example, there are in total 566 API methods in *mockito* and 344 API methods in *jMock*, but only 16 API pairs between the two libraries are covered in the ground truth. We would like to ensure that our automatic evaluation is not biased by the small number of API mappings involved in code changes. For the 128 query APIs (no overlapping with the ground-truth APIs) in the four pairs of libraries in the ground truth, the MRR, Re@1, Re@5 and Re@10 for the recommendation results are 0.516, 0.408, 0.603 and 0.68 respectively. Overall, our manual evaluation results are consistent with automatic evaluation results based on a different set of ground-truth API mappings labeled by other researchers.

The MRR of manual evaluation is slightly lower than that of automatic evaluation (MRR=0.556, see Table. 6). The Re@1 is similar to that of automatic evaluation (0.409). The Re@5 and Re@10 are relative lower than those of automatic evaluation. A key difference between the APIs in the ground-truth API mappings and the randomly sampled 80 APIs is that all APIs in the ground-truth are confirmed to have analogical APIs, while randomly sampled 80 APIs likely do not have analogical APIs. For example, APIs for a unique feature in one library essentially do not have counterparts in the other library. Therefore, our automatic evaluation shows that if an analogical API does exist, our approach can highly likely rank it in the top 5 or 10 recommended APIs. However, our approach will fail when there are fundamentally no analogical APIs to recommend. In such cases, Re@5 or Re@10 will be zero, resulting in relative lower performance in manual evaluation. We will elaborate further on this point in Section 5.4.3.

TABLE 7
Manual evaluation of analogical API recommendation in 12 pairs of libraries with diverse functionalities

Analogical library pair	Functionality	MRR	Re@1	Re@5	Re@10
Apache IO → Guava IO	I/O	0.538	0.38	0.69	0.75
Guava Base → Apache Lang	Utility methods	0.588	0.5	0.69	0.72
JSON → gson	(de)serialization	0.521	0.44	0.59	0.75
mockito → jMock	mocking	0.416	0.31	0.44	0.5
dom4j → jdom	XML parsing	0.512	0.38	0.63	0.72
junit → testng	unit testing	0.658	0.63	0.69	0.72
log4j → slf4j	logging	0.473	0.41	0.41	0.53
jexcelapi → apache-poi	excel manipulation	0.611	0.47	0.75	0.84
pdfbox → itext	PDF manipulation	0.404	0.28	0.47	0.53
lucene → solr	information retrieval	0.469	0.38	0.47	0.47
charts4j → jfreechart	visualization	0.315	0.16	0.44	0.53
opennlp → stanford-nlp	NLP	0.453	0.34	0.53	0.59

5.4.3 Mining Analogical APIs in Diverse Libraries

The four pairs of libraries in the ground-truth cover only limited functionalities (see Table 2). We would like to extend our evaluation to other libraries with different functionalities. We select 8 more pairs of analogical libraries from our dataset. Table 7 shows the manual evaluation results for these additional 8 pairs of libraries as well as the four pairs of libraries in the ground-truth. The Cohen's Kappa metric among annotator's decisions is 93.6% which indicates almost perfect agreement [66].

We can see that the performance may vary for libraries with different functionalities. Overall, the performance is good for libraries from application domains that have a common set of application logic, such as (*Guava Base, Apache Commons Lang*) for utility methods, (*junit, testng*) for unit testing, (*jexcelapi, apache-poi*) for spreadsheet manipulation, (*dom4j, jdom*) for XML parsing and manipulation. These analogical libraries tend to have analogical APIs for the relevant application logic. We then analyze the potential effects from different factors including the lib uniqueness, lib populairity, and API design granularity.

Influence of the lib uniqueness: Analogical libraries often have different focuses, even they are developed for the same application domain. For example, both *itext* and *pdfbox* can be used for PDF manipulation, but *itext* focuses more on PDF generation and modification, while *PDFBox* is a PDF content extraction library with basic modification functionality [67]. As another example, *chart4j* focuses on wrapping Google Chart API in Java, while *JFreeChart* is a traditional Java chart library [68]. For such analogical libraries, if one library has some unique features that do not have counterparts in the other library, our approach will fail for these unique features as there are no counterparts to recommend.

Influence of the lib popularity: Table 8 shows that the MRR, Re@1, Re@5, Re@10 for the unpopular library pairs is the highest, following with the popular pairs and hybrid pairs. It seems that if libraries are with the similar popularity in the pair, out model can gain better performance. We further adopt the Pearson correlation [69] to analyze the detailed correlation between the model performance and the library occurrence frequency in Stack Overflow. The correlation coefficient value between MRR, Re@1, Re@5, Re@10 and lib frequency is ranging from -0.11 to -0.16 with the p-value ranging from 0.63 to 0.74. It shows that there is no significant correlation between the popularity of libraries and the performance. There are three kinds of information in this study used for encoding API similarity. The API docu-

TABLE 8
The performance of our model in different popularity group

Analogical library pair	MRR	Re@1	Re@5	Re@10
Popular	0.501	0.425	0.51	0.563
Hybrid	0.463	0.344	0.55	0.642
Unpopular	0.546	0.42	0.67	0.73

mentation and name is not influenced by the lib popularity, but only API usage embedding may be influenced as our API embedding model is sensitive to API usage frequency to some extent. It accounts for the phenomenon that libraries in the pair with similar popularity have better results that the hybrid groups. But as the API embedding only accounts for 30% weight of API similarity inference, the popularity influence is not significant for the overall performance.

Influence of the API design granularity: Often, the API design of analogical libraries is rather different even for the similar features. Sometimes, one library intends to provide a facade for the other library, for example, *solr* for *lucene* and *slf4j* for *log4j*. A facade often provides an easier or simpler interface to the underlying library. In other cases, analogical libraries may support different practices and their APIs are at different level of granularity. For example, the two test mocking libraries *mockito* and *jMock* differ a lot in the way they deal with expectations [70]. Such design differences will result in complex (e.g., one-to-many or many-to-many) API mappings for the same feature. This leads to difficulty in determining analogical APIs between libraries.

The analysis of different factors show that the results are influenced by the nature of the library such as the library uniqueness and its API design granularity, instead of its popularity.

5.5 Threats to Validity

One threat to internal validity is the availability of API usage and document data for adopting our approach, especially for unpopular libraries. To understand the availability of the required document data, we first take all java libraries from our previous work [3] which recommends analogical third-party libraries across different programming languages. Among 1805 java libraries extracted from tags in Stack Overflow, we randomly sample 50 of them to check if they have corresponding Javadoc. We also count these libraries occurrence in Stack Overflow tags, and their tag frequency ranges from 3 (unpopular) to 138049 (very popular) as of Aug 28, 2018. The results show that 49 (98%) of them provide detailed Javadoc[7]. Only 1 library rest.li[8] does not have explicit Javadoc website, but it is easy to extract the Javadoc from their source code[9]. It demonstrates that the data such as API usage, method Javadocs, API specification is generally available, and can be extracted from source code and API's official sites with reasonable amount of engineering effort and time.

The second threat to internal validity is the optimization of several approach parameters. Our experiments show that our approach is reliable for a range of parameter settings. The third internal threat is that the collected GitHub projects for training the API embedding model and the skip-thought model may be of low quality. To filter out the potential low-quality projects, we have removed projects whose star number is less than 10. The fourth threat to internal validity is the scope of the API usage. It is well-known that it is difficult to define an API's usage scope, but we do not explicitly take the API usage scope into consideration in this work. Instead, we encode the API usage semantic by its neighbour API calls in the window. As each method contain 5.3 API calls in average, we set up the window size as 5 so that we can include most potentially related APIs. Furthermore, note that continuous skip-gram model does not consider the ordering of words in the context window. Of course, it is likely that some API calls within the context window may not be in the target APIs usage scope. However, the word embedding technique relies on overall co-occurrence statistics to lower their weight of less frequent API co-occurrence in the training data, i.e., the less related API appears less frequently in the window, also contributing less to the final embedding. In addition, to further mitigate the potential bias of encoding API usage pattern, we also encode the API names and API documents to jointly represent the semantic of the API. Another internal threat is that we do not take into account inter-procedural calls. This may may negatively influence extracting analogical APIs that usually happen only in the same code block (method), and not across different methods. But note that apart from API usage pattern, we also incorporate the information of API name and API comments which may mitigate the potential negative influence. And we will improve it in the future [71], [72].

The last internal threat is that we only examine a small number of analogical APIs due to the significant effort required. But our manual evaluation involves 3 times more APIs than Teyton's dataset [16], and the results are consistent with the performance of our approach on the ground-truth APIs identified in Teyton's dataset. In fact, the difficulty in manually building a big dataset of analogical APIs actually calls for some unsupervised method like ours to recommend analogical APIs for third-party libraries. We build an online application to release our analogical APIs database for public access. The power of the crowd could be exploited to examine the results in large scale. As more ground-truth mappings are accumulated from the online application, they could be used to further optimize system parameters.

A threat to external validity includes the generalization of our findings for more analogical libraries and for other programming languages. In this work, we examine 12 pairs of analogical libraries with diverse functionalities. Although our approach works reliably in general, we observe the performance differences due to the characteristics of different libraries. We release our analogical API database via an online application. We hope the power of the crowd could be exploited to examine more results in large scale such as the clicking rates. As more ground-truth mappings are accumulated, they could be used to further optimize approach parameters. Although programming languages differ from each other, they generally have comments, API sequences and certain format of method naming convention. Therefore, by customizing the implementation details

7. The detailed results of this pilot study can be seen in https://similarapi.appspot.com/libraryJavadoc.html
8. https://github.com/linkedin/rest.li/wiki/Rest.li-User-Guide
9. https://github.com/linkedin/rest.li

of data preprocessing steps, our approach can potentially be extended to other languages.

6 Related Work

There are three types of software migration tasks: across version migration [73], language/platform migration, and library migration. Typical techniques to support across version migration, such as Diff-Catchup [74], analyze already-migrated code (e.g., unit tests, demo applications) to infer code change patterns to update a software application to work with the new version of the framework. Existing techniques [8], [10] supporting language/platform migration require the same software implemented in different languages or platforms, for example, the original Lucene written in Java and the Lucene.NET written in C#. In contrast, our model is totally unsupervised without relying on such parallel corpus. Gokhale et al. [9] relieves such parallel-implementations requirement, but it still requires functionality similar applications for different platforms, e.g., the TicTacToe game for JavaME and Android developed by different projects.

Some techniques [16], [75] for library migration adopt the similar approach to across version and language/platform migration. For example, Teyton et al. [16] infer likely API mappings between similar libraries by examining already-ported code, i.e., changes made to port an application from using one library's APIs to another library's. It is relatively easy to collect already-migrated code across framework versions or functionality similar applications across languages/platforms, but it is unlikely to have an already-ported application for an arbitrary pair of analogical libraries. Similar to Gokhale et al.'s work [9], Santhiar et al. [75] mine similar units tests for migrating math APIs. Collecting similar unit tests for a few Math libraries is feasible, but it is difficult to extend this method to the wide range of analogical libraries that we are concerned with. Chen et al. [3], [76] extract analogical third-party libraries across different programming languages by incorporating relational, categorical and semantic information from Stack Overflow. Different from those works about mining analogical libraries, this study focuses on more fine-grained similar API extraction. Apart from skip-gram model, we adopt more advanced algorithm (skip-thoughts model) to encode API name and documentation similarity which is more complicated and difficult to embed than tags in Stack Overflow.

To eliminate this need for already migrated or functionality similar code, Pandita et al. [11] adopt text mining to identify likely API mappings based on the textual similarity of API names and documents. But they use traditional TF-IDF based similarity metric which cannot properly handle lexical gaps in API descriptions from different libraries. In contrast, we adopt unsupervised RNN model to overcome the lexical gap issue in API descriptions. Lu et al. [77] and Gu et al. [59] infer similar APIs across different programming languages from API documents. Different from their research, our approach incorporates more information such as API usage, and API names for inferring analogical APIs and our model is completely unsupervised.

Much work has been done on mining API usage patterns [71], [78], [79] and recommending API usage examples [80], [81]. Their focus is on APIs that are frequently used together, while our focus is on analogical APIs between libraries. Analogical APIs may be frequently used with some other common APIs, but they are unlikely used together. Furthermore, our approach does not need to explicitly mine what APIs are often used with analogical APIs, because it uses unsupervised word embeddings to embed the knowledge of surrounding APIs. Some research works [60], [82], [83] investigates several applications of API embeddings. They also adopt word2vec model [12] to embed API call sequences, but they use attributional similarity of API embeddings to determine related APIs within one language or across two languages. For the one-language scenario, our Baseline1 in Section 5.3.1 is the same as Nguyen's method, and our relational similarity of API-library embeddings outperforms it. For the across-two-languages scenario, Nguyen's method requires a corpus of known mapping samples (e.g., Lucene in Java versus C#) to compute the transformation matrix between the API embeddings of Java and C# (as seen in the 1st and 3rd sentences of Section V.C [60]). Our work deals with 111 pairs of analogical libraries for which no such known mapping samples exist. Therefore, Nguyen's method cannot be applied in our analogical-libraries setting. RNN models have also been used for recommending API calls [58] or detecting code clones [84]. Different from our use of unsupervised RNN model to embed API name/document semantics, these works require labelled data for supervised model training, for example method-comment pairs or method-AST pairs.

7 Conclusion & Future Work

This paper presents a novel unsupervised deep learning approach for inferring likely analogical API mappings between third-party libraries. Our approach applies word embedding techniques to API call sequences, and thus eliminates the need for already-ported or functionality similar applications for reasoning API usage semantics. It adopts skip-thoughts model, an unsupervised RNN model for quantifying API description similarity in the presence of lexical gap. Our approach is the first attempt to incorporate all API usage, name and document similarities for inferring likely analogical APIs of third-party libraries. Our evaluation shows that the three kinds of API similarities are complementary to each other, and as a whole they can reliably recommend likely analogical APIs for libraries with diverse functionalities. To evaluate our approach, we build the largest ever database of analogical APIs for 583,501 APIs of 111 pairs of analogical Java libraries. In the future, we will extend our work to one-to-many and many-to-many API mappings which complements our current one-to-one mapping. We will also exploit the power of the crowd who visits our web application to further evaluate and improve our approach.

References

[1] F. Thung, L. David, and J. Lawall, "Automated library recommendation," in *2013 20th Working Conference on Reverse Engineering (WCRE 2013): Proceedings: Koblenz, Germany, 14-17 October 2013*, 2013, pp. 182–191.

[2] C. Teyton, J.-R. Falleri, and X. Blanc, "Mining library migration graphs," in *2012 19th Working Conference on Reverse Engineering*. IEEE, 2012, pp. 289–298.

[3] C. Chen, S. Gao, and Z. Xing, "Mining analogical libraries in q&a discussions–incorporating relational and categorical knowledge into word embedding," in *2016 IEEE 23rd International Conference on Software Analysis, Evolution, and Reengineering (SANER)*, vol. 1. IEEE, 2016, pp. 338–348.

[4] "Windows phone interoperability: Windows phone api mapping," http://windowsphone.interoperabilitybridges.com/porting, 2017.

[5] E. Duala-Ekoko and M. P. Robillard, "Asking and answering questions about unfamiliar apis: an exploratory study," in *Proceedings of the 34th International Conference on Software Engineering*. IEEE Press, 2012, pp. 266–276.

[6] C. Chen, Z. Xing, and L. Han, "Techland: Assisting technology landscape inquiries with insights from stack overflow," in *Software Maintenance and Evolution (ICSME), 2016 IEEE International Conference on*. IEEE, 2016, pp. 356–366.

[7] C. Chen and Z. Xing, "Mining technology landscape from stack overflow," in *Proceedings of the 10th ACM/IEEE International Symposium on Empirical Software Engineering and Measurement*. ACM, 2016, p. 14.

[8] H. Zhong, S. Thummalapenta, T. Xie, L. Zhang, and Q. Wang, "Mining api mapping for language migration," in *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering-Volume 1*. ACM, 2010, pp. 195–204.

[9] A. Gokhale, V. Ganapathy, and Y. Padmanaban, "Inferring likely mappings between apis," in *2013 35th International Conference on Software Engineering (ICSE)*. IEEE, 2013, pp. 82–91.

[10] A. T. Nguyen, H. A. Nguyen, T. T. Nguyen, and T. N. Nguyen, "Statistical learning approach for mining api usage mappings for code migration," in *Proceedings of the 29th ACM/IEEE international conference on Automated software engineering*. ACM, 2014, pp. 457–468.

[11] 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*. IEEE, 2015, pp. 231–240.

[12] T. Mikolov, K. Chen, G. Corrado, and J. Dean, "Efficient estimation of word representations in vector space," *arXiv preprint arXiv:1301.3781*, 2013.

[13] 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*, 2013, pp. 3111–3119.

[14] R. Kiros, Y. Zhu, R. R. Salakhutdinov, R. Zemel, R. Urtasun, A. Torralba, and S. Fidler, "Skip-thought vectors," in *Advances in neural information processing systems*, 2015, pp. 3294–3302.

[15] C. Chen, Z. Xing, and Y. Liu, "What's spain's paris? mining analogical libraries from q&a discussions," *Empirical Software Engineering*, pp. 1–40, 2018.

[16] C. Teyton, J.-R. Falleri, and X. Blanc, "Automatic discovery of function mappings between similar libraries." in *WCRE*, 2013, pp. 192–201.

[17] Y. Liu, Z. Liu, T.-S. Chua, and M. Sun, "Topical word embeddings." in *AAAI*, 2015, pp. 2418–2424.

[18] C. De Boom, S. Van Canneyt, S. Bohez, T. Demeester, and B. Dhoedt, "Learning semantic similarity for very short texts," in *Data Mining Workshop (ICDMW), 2015 IEEE International Conference on*. IEEE, 2015, pp. 1229–1234.

[19] G. Chen, C. Chen, Z. Xing, and B. Xu, "Learning a dual-language vector space for domain-specific cross-lingual question retrieval," in *Automated Software Engineering (ASE), 2016 31st IEEE/ACM International Conference on*. IEEE, 2016, pp. 744–755.

[20] C. Chen, Z. Xing, and X. Wang, "Unsupervised software-specific morphological forms inference from informal discussions," in *Proceedings of the 39th International Conference on Software Engineering*. IEEE Press, 2017, pp. 450–461.

[21] Y. Huang, C. Chen, Z. Xing, T. Lin, and Y. Liu, "Tell them apart: distilling technology differences from crowd-scale comparison discussions," in *Proceedings of the 33rd ACM/IEEE international conference on automated software engineering. ACM*, 2018, pp. 214–224.

[22] T. Mikolov, M. Karafiát, L. Burget, J. Cernockỳ, and S. Khudanpur, "Recurrent neural network based language model." in *Interspeech*, vol. 2, 2010, p. 3.

[23] A. Graves, A.-r. Mohamed, and G. Hinton, "Speech recognition with deep recurrent neural networks," in *2013 IEEE international conference on acoustics, speech and signal processing*. IEEE, 2013, pp. 6645–6649.

[24] M. White, C. Vendome, M. Linares-Vásquez, and D. Poshyvanyk, "Toward deep learning software repositories," in *2015 IEEE/ACM 12th Working Conference on Mining Software Repositories*. IEEE, 2015, pp. 334–345.

[25] K. Cho, B. Van Merriënboer, C. Gulcehre, D. Bahdanau, F. Bougares, H. Schwenk, and Y. Bengio, "Learning phrase representations using rnn encoder-decoder for statistical machine translation," *arXiv preprint arXiv:1406.1078*, 2014.

[26] C. Chen, Z. Xing, and Y. Liu, "By the community & for the community: a deep learning approach to assist collaborative editing in q&a sites," *Proceedings of the ACM on Human-Computer Interaction*, vol. 1, no. CSCW, pp. 32:1–32:21, 2017.

[27] S. Gao, C. Chen, Z. Xing, Y. Ma, W. Song, and S.-W. Lin, "A neural model for method name generation from functional description," in *2019 IEEE 26th International Conference on Software Analysis, Evolution, and Reengineering (SANER)*. IEEE, 2019.

[28] B. Dagenais and L. Hendren, "Enabling static analysis for partial java programs," in *ACM Sigplan Notices*, vol. 43, no. 10. ACM, 2008, pp. 313–328.

[29] R. Miller and R. Kasparian, *Java for artists: the art, philosophy, and science of object-oriented programming*. Page 220, 2006.

[30] G. Palmer, *Technical Java: developing scientific and engineering applications*. Page 126, 2003.

[31] R. S. Grover, *Programming with Java: A Multimedia Approach*. Page 215, 2011.

[32] I. Jolliffe, *Principal component analysis*. Wiley Online Library, 2002.

[33] A. Hindle, E. T. Barr, Z. Su, M. Gabel, and P. Devanbu, "On the naturalness of software," in *2012 34th International Conference on Software Engineering (ICSE)*. IEEE, 2012, pp. 837–847.

[34] Oracle. (2017) How to Write Doc Comments for the Javadoc Tool. [Online]. Available: http://www.oracle.com/technetwork/articles/java/index-137868.html

[35] M. Allamanis and C. Sutton, "Mining idioms from source code," in *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*. ACM, 2014, pp. 472–483.

[36] M. Allamanis, E. T. Barr, C. Bird, and C. Sutton, "Suggesting accurate method and class names," in *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*. ACM, 2015, pp. 38–49.

[37] V. Raychev, M. Vechev, and A. Krause, "Predicting program properties from big code," in *ACM SIGPLAN Notices*, vol. 50, no. 1. ACM, 2015, pp. 111–124.

[38] R. Dyer, H. A. Nguyen, H. Rajan, and T. N. Nguyen, "Boa: A language and infrastructure for analyzing ultra-large-scale software repositories," in *Proceedings of the 2013 International Conference on Software Engineering*. IEEE Press, 2013, pp. 422–431.

[39] ——, "Boa: Ultra-large-scale software repository and source-code mining," *ACM Transactions on Software Engineering and Methodology (TOSEM)*, vol. 25, no. 1, p. 7, 2015.

[40] R. Dyer, H. Rajan, H. A. Nguyen, and T. N. Nguyen, "Mining billions of ast nodes to study actual and potential usage of java language features," in *Proceedings of the 36th International Conference on Software Engineering*. ACM, 2014, pp. 779–790.

[41] L. Xavier, A. Brito, A. Hora, and M. T. Valente, "Historical and impact analysis of api breaking changes: A large-scale study," in *Software Analysis, Evolution and Reengineering (SANER), 2017 IEEE 24th International Conference on*. IEEE, 2017, pp. 138–147.

[42] T. Zhang, G. Upadhyaya, A. Reinhardt, H. Rajan, and M. Kim, "Are code examples on an online q&a forum reliable?: a study of api misuse on stack overflow," in *Proceedings of the 40th International Conference on Software Engineering*. ACM, 2018, pp. 886–896.

[43] "Eclipse jdt," http://www.eclipse.org/jdt/, 2017.

[44] "Apache commons io," http://commons.apache.org/proper/commons-io/, 2017.

[45] "Guava common io," https://github.com/google/guava/tree/master/guava/src/com/google/common/io, 2017.

[46] "Apache commons lang," https://commons.apache.org/proper/commons-lang/, 2017.

[47] "Guava common base," https://github.com/google/guava/tree/master/guava/src/com/google/common/base, 2017.

[48] "Json," https://github.com/stleary/JSON-java, 2017.

[49] "Google-gson," https://github.com/google/gson, 2017.

[50] "Mockito," http://site.mockito.org/, 2017.

[51] "Jmock," http://www.jmock.org/, 2017.
[52] C. Sun, D. Lo, S.-C. Khoo, and J. Jiang, "Towards more accurate retrieval of duplicate bug reports," in *Proceedings of the 2011 26th IEEE/ACM International Conference on Automated Software Engineering*. IEEE Computer Society, 2011, pp. 253–262.
[53] X. Wang, L. Zhang, T. Xie, J. Anvik, and J. Sun, "An approach to detecting duplicate bug reports using natural language and execution information," in *Proceedings of the 30th international conference on Software engineering*. ACM, 2008, pp. 461–470.
[54] M. M. Rahman, C. K. Roy, and D. Lo, "Rack: Automatic api recommendation using crowdsourced knowledge," in *Software Analysis, Evolution, and Reengineering (SANER), 2016 IEEE 23rd International Conference on*, vol. 1. IEEE, 2016, pp. 349–359.
[55] Q. Huang, X. Xia, Z. Xing, D. Lo, and X. Wang, "Api method recommendation without worrying about the task-api knowledge gap," in *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering*. ACM, 2018, pp. 293–304.
[56] J. Zhang, H. Jiang, Z. Ren, and X. Chen, "Recommending apis for api related questions in stack overflow," *IEEE Access*, vol. 6, pp. 6205–6219, 2018.
[57] X. Ye, H. Shen, X. Ma, R. Bunescu, and C. Liu, "From word embeddings to document similarities for improved information retrieval in software engineering," in *Proceedings of the 38th International Conference on Software Engineering*. ACM, 2016, pp. 404–415.
[58] X. Gu, H. Zhang, D. Zhang, and S. Kim, "Deep api learning," in *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*. ACM, 2016, pp. 631–642.
[59] ——, "Deepam: Migrate apis with multi-modal sequence to sequence learning," *arXiv preprint arXiv:1704.07734*, 2017.
[60] T. D. Nguyen, A. T. Nguyen, H. D. Phan, and T. N. Nguyen, "Exploring api embedding for api usages and applications," in *Proceedings of the 39th International Conference on Software Engineering*. IEEE Press, 2017, pp. 438–449.
[61] H. P. Luhn, "A statistical approach to mechanized encoding and searching of literary information," *IBM Journal of research and development*, vol. 1, no. 4, pp. 309–317, 1957.
[62] Q. V. Le and T. Mikolov, "Distributed representations of sentences and documents." in *ICML*, vol. 14, 2014, pp. 1188–1196.
[63] S. Arora, Y. Liang, and T. Ma, "A simple but tough-to-beat baseline for sentence embeddings," in *Proceedings of the 5th International Conference on Learning Representations (ICLR)*, 2017.
[64] R. Henao, C. Li, L. Carin, Q. Su, D. Shen, G. Wang, W. Wang, M. R. Min, and Y. Zhang, "Baseline needs more love: On simple word-embedding-based models and associated pooling mechanisms," in *Proceedings of the 56th Annual Meeting of the Association for Computational Linguistics, ACL 2018, Melbourne, Australia, July 15-20, 2018, Volume 1: Long Papers*, 2018, pp. 440–450.
[65] R. Singh and N. S. Mangat, *Elements of survey sampling*. Springer Science & Business Media, 2013, vol. 15.
[66] J. R. Landis and G. G. Koch, "The measurement of observer agreement for categorical data," *biometrics*, pp. 159–174, 1977.
[67] (2017) itext: Comparison with pdfbox. [Online]. Available: http://itext.2136553.n4.nabble.com/Comparison-with-PDFBox-td2140747.html
[68] (2017) jfreecharts vs charts4j. [Online]. Available: https://stackoverflow.com/questions/1639282/\#1639446
[69] K. Pearson, "Note on regression and inheritance in the case of two parents," *Proceedings of the Royal Society of London*, vol. 58, pp. 240–242, 1895.
[70] (2017) Unit testing with mocks - easymock, jmock and mockito. [Online]. Available: http://blogs.justenougharchitecture.com/unit-testing-with-mocks-easymock-jmock-and-mockito/
[71] 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 28th IEEE/ACM International Conference on Automated Software Engineering*. IEEE Press, 2013, pp. 646–651.
[72] X. Gu, H. Zhang, and S. Kim, "Deep code search," in *Proceedings of the 40th International Conference on Software Engineering*. ACM, 2018, pp. 933–944.
[73] C. Chen and Z. Xing, "Towards correlating search on google and asking on stack overflow," in *Computer Software and Applications Conference (COMPSAC), 2016 IEEE 40th Annual*, vol. 1. IEEE, 2016, pp. 83–92.
[74] Z. Xing and E. Stroulia, "Api-evolution support with diff-catchup," *IEEE Transactions on Software Engineering*, vol. 33, no. 12, pp. 818–836, 2007.
[75] A. Santhiar, O. Pandita, and A. Kanade, "Mining unit tests for discovery and migration of math apis," *ACM Transactions on Software Engineering and Methodology (TOSEM)*, vol. 24, no. 1, p. 4, 2014.
[76] C. Chen and Z. Xing, "Similartech: automatically recommend analogical libraries across different programming languages," in *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering*. ACM, 2016, pp. 834–839.
[77] Y. Lu, G. Li, Z. Zhao, L. Wen, and Z. Jin, "Learning to infer api mappings from api documents," in *International Conference on Knowledge Science, Engineering and Management*. Springer, 2017, pp. 237–248.
[78] H. Zhong, T. Xie, L. Zhang, J. Pei, and H. Mei, "Mapo: Mining and recommending api usage patterns," *ECOOP 2009–Object-Oriented Programming*, pp. 318–343, 2009.
[79] H. Zhong and H. Mei, "An empirical study on api usages," *IEEE Transactions on Software Engineering*, 2017.
[80] L. Moreno, G. Bavota, M. Di Penta, R. Oliveto, and A. Marcus, "How can i use this method?" in *Software Engineering (ICSE), 2015 IEEE/ACM 37th IEEE International Conference on*, vol. 1. IEEE, 2015, pp. 880–890.
[81] A. T. Nguyen, M. Hilton, M. Codoban, H. A. Nguyen, L. Mast, E. Rademacher, T. N. Nguyen, and D. Dig, "Api code recommendation using statistical learning from fine-grained changes," in *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*. ACM, 2016, pp. 511–522.
[82] H. D. Phan, A. T. Nguyen, T. D. Nguyen, and T. N. Nguyen, "Statistical migration of api usages," in *Software Engineering Companion (ICSE-C), 2017 IEEE/ACM 39th International Conference on*. IEEE, 2017, pp. 47–50.
[83] Y. Lu, G. Li, R. Miao, and Z. Jin, "Learning embeddings of api tokens to facilitate deep learning based program processing," in *Knowledge Science, Engineering and Management*, F. Lehner and N. Fteimi, Eds. Springer International Publishing, 2016, pp. 527–539.
[84] M. White, M. Tufano, C. Vendome, and D. Poshyvanyk, "Deep learning code fragments for code clone detection," in *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering*. ACM, 2016, pp. 87–98.