Easy-to-Deploy API Extraction by Multi-Level Feature Embedding and Transfer Learning

Suyu Ma, Zhenchang Xing, Chunyang Chen, Cheng Chen, Lizhen Qu, Guoqiang Li

Abstract—Application Programming Interfaces (APIs) have been widely discussed on social-technical platforms (e.g., Stack Overflow). Extracting API mentions from such informal software texts is the prerequisite for API-centric search and summarization of programming knowledge. Machine learning based API extraction has demonstrated superior performance than rule-based methods in informal software texts that lack consistent writing forms and annotations. However, machine learning based methods have a significant overhead in preparing training data and effective features. In this paper, we propose a multi-layer neural network based architecture for API extraction. Our architecture automatically learns character-, word- and sentence-level features from the input texts, thus removing the need for manual feature engineering and the dependence on advanced features (e.g., API gazetteers) beyond the input texts. We also propose to adopt transfer learning to adapt a source-library-trained model to a target-library, thus reducing the overhead of manual training-data labeling when the software text of multiple programming languages and libraries need to be processed. We conduct extensive experiments with six libraries of four programming languages which support diverse functionalities and have different API-naming and API-mention characteristics. Our experiments investigate the performance of our neural architecture for API extraction in informal software texts, the importance of different features, the effectiveness of transfer learning. Our results confirm not only the superior performance of our neural architecture than existing machine learning based methods for API extraction in informal software texts, but also the easy-to-deploy characteristic of our neural architecture.

Index Terms—API extraction, CNN, Word embedding, LSTM, Transfer learning

1 Introduction

Application Programming Interfaces (APIs) are a set of definitions, functions and modules for developing software programs. To support the use of APIs and solve the usage issues, developers not only create formal API specifications and tutorials (e.g., Java API, Android Developers), but also generate large numbers of informal discussions on APIs (e.g., Stack Overflow questions and answers) [1]. Distinguishing API mentions from general natural language words in API documentation is referred to as API extraction or API recognition in the literature [2]. Fig. 1 illustrates an example of API extraction in natural language sentences. API extraction is crucial for many downstream applications. For traceability recovery across software documents, API extraction lays the foundation of linking code-like terms to specific code elements in an API or API documentation [3], [4]. For entity-centric search, API extraction can be exploited to create a thesaurus of software-specific terms and commonly used morphological forms [5], [6], build an API caveats knowledge graph [7] and search for appropriate APIs for programming tasks [8]. For domain-specific question answering, API extraction can help select answer paragraphs and generate useful answer summary [9].

- *Suyu Ma, Chunyang Chen (corresponding author) and Lizhen Qu are with Faculty of Information Technology, Monash University, Australia. E-mail: masuyu2015@outlook.com, chunyang.chen@monash.edu, Lizhen.Qu@monash.edu.*
- *Zhenchang Xing is with College of Engineering & Computer Science, Australian National University, Australia. E-mail: zhenchang.xing@anu.edu.au*
- *Cheng Chen is with PricewaterhouseCoopers Firm, China. E-mail: cc94226@live.com*
- *Guoqiang Li (corresponding author) is with School of Software, Shanghai Jiao Tong University, China. E-mail: li-gq@cs.sjtu.edu.cn*

Manuscript received October 31, 2018; revised August 24, 2019.

Unlike formal API documentation where API mentions are consistently written and annotated, API mentions in informal software texts usually lack consistent writing forms and annotations [2]. For example, the methods *apply* and *bfill* in Fig. 1 are not mentioned in their fully qualified name and are not annotated with a special tag like <code>. Furthermore, an API may have a common-word simple name (e.g., *apply*, *series*). Our analysis of API simple names in six libraries of four programming languages reveals that 6% to 66% (median 42%) APIs of these libraries have common-word simple name. Such APIs are referred to as polysemous APIs [2], because mentioning them in their simple name without special tag creates a common-word polysemy issue for API extraction [2].

Polysemous API mentions, together with other informality of API mentions, render rule-based extraction of API mentions unreliable for informal software texts. Recently, several machine learning based API extraction methods [2], [10] have been proposed. These machine learning based methods have demonstrated superior performance for API extraction in informal texts than rule-based methods. However, a major barrier for deploying such machine learning based methods is the significant overhead required for manual labeling of model training data and manual feature engineering.

API extraction in informal software texts can be regarded as a Named Entity Recognition (NER) task [11] in Natural Language Processing (NLP). An NER task in general English text detects mentions of named entities, such as people, locations and organizations. It deals with single language. However, API extraction has to deal with hundreds of libraries and frameworks discussed by developers. Training a reliable machine learning based API extraction model for a

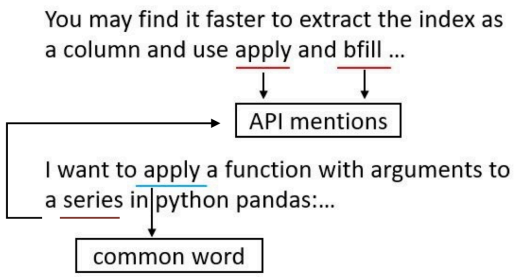

Fig. 1: Illustrating API Extraction Task

library often requires several hundreds of manually labeled sentences mentioning this library's APIs [2]. The effort to prepare training data for hundreds of libraries would be prohibitive. Furthermore, it may also be difficult to prepare sufficient high-quality training data for APIs of some less frequently discussed libraries or frameworks.

Another related challenge is to select effective features for a machine learning model to recognize a particular library's APIs. Although developers follow general naming conventions, orthographic features of APIs still vary greatly from one library to another. For example, as reported in Section 2, different libraries have different percentages of polysemous API names. Furthermore, users of some libraries tend to mention APIs with clear orthographic features (e.g., package names, bracket and/or dot), while users of other libraries tend to directly mention API simple names. Functionalities of software libraries also vary greatly, such as graphical user interface, numeric computation, machine learning, data visualization, database access. As such, discussion contexts of a library's APIs, like *Pandas* (a Python machine learning library), often differ from those of another library's APIs, like *JDBC* (a Java database access library).

Designers of a machine learning based API extraction model have to manually select the most effective features for different libraries' APIs. This is a challenging task as there are dozens of features to choose from[1]. Unsupervised word embeddings have been explored for API extraction tasks [2], but there has been no work on exploiting character- and sentence-context embeddings from input texts for API extraction. Furthermore, some advanced features to boost API extraction performance, such as word clusters and API gazetteers, have to be hand-crafted.Without such advanced features, existing machine learning based API extraction methods perform poorly using only orthographic features from the input texts [2].

The easy deployment is defined as that the model can be easily trained for different datasets, without requiring any manual feature engineering. To make machine learning based API extraction methods easy to deploy in practice, we must reduce the overhead of preparing training data and effective features, and remove the dependence on additional features beyond input texts. In this paper, we design a neural architecture for API extraction in informal software text. Our neural architecture is composed of the character-level convolutional neural network (CNN), word-level embeddings, and sentence-level Bi-directional Long Short-Term Memory (Bi-LSTM) network for automatically learning character-, word- and sentence-level features from input texts, respectively. This neural architecture can be trained in an end-to-end fashion, thus removing the need for manual feature engineering and the need for additional features beyond input texts, and greatly reducing the amount of new training data needed for adapting a model to different libraries.

Furthermore, our analysis of the API-naming and API-mention characteristics suggests that the characteristics of API names, API mentions and discussion contexts differ across libraries, but they also share certain level of commonalities. To exploit such commonalities for easy deployment of API extraction model, we adopt transfer learning [12], [13] to fine-tune a model trained with a source library's API discussion texts to a target library. This helps reduce the amount of training data required for training a high-quality target-library model, compared with training the model from scratch with randomly initialized model parameters. The design of our multi-level neural architecture enables the fine-tuning of different levels of features in transfer learning.

We conduct extensive experiments to evaluate the performance of the proposed neural architecture for API extraction as well as the effectiveness of transfer learning. Our experiments involve three Python libraries (*Pandas*, *NumPy* and *Matplotlib*), one Java library (*JDBC*), one JavaScript library (*React*), and one C library (*OpenGL*). As discussed in Section 2, these six libraries support diverse functionalities and have different API-naming, API-mention and discussion-context characteristics. We manually label API mentions in 3600 Stack Overflow posts (600 for each library) for the experiments. Our experiments confirm the effectiveness of our neural architecture in learning multi-level features from the input texts, and show that the learned features can support high-quality API extraction in informal software texts, without the need for additional hand-crafted features beyond the input texts. Our experiments also confirm the effectiveness of transfer learning [14] in boosting the target-library model performance with much less training data, even in few-shot (about 10 posts) training settings.

This paper makes the following four contributions:

- Our work is the first one to consider not only the performance of machine learning based API extraction methods but also the easy deployment of such methods for the software text of multiple programming languages and libraries.
- We propose a multi-layer neural architecture to automatically learn to extract effective features from the input texts for API extraction, thus removing the need for manual feature engineering as well as the dependence on features beyond the input texts.
- We adopt transfer learning to reduce the overhead of manual labeling of the training data of a subject library. We evaluate the effectiveness of transfer learning across libraries and programming languages and analyze the factors that affect its effectiveness.
- We conduct extensive experiments to evaluate our architecture as a whole as well its components. Our results reveal insights into the design of effective mechanisms for API extraction tasks.

1. https://nlp.stanford.edu/nlp/javadoc/javanlp/edu/stanford/nlp/ie/NERFeatureFactory.html

TABLE 1: Statistics of Polysemous APIs

Library	APIs	Polysemous APIs	Percentage
Matplotlib	3877	622	16.04%
Pandas	774	426	55.04%
Numpy	2217	917	41.36%
Opengl	850	52	6.12%
React	238	157	65.97%
JDBC	1468	633	43.12%

TABLE 2: Statistics of API-Mention Characteristics

Library	Orthographic	Long tokens	Common context words
Matplotlib	62.38%	21.56%	20.64%
Pandas	67.11%	32.22%	34.23%
Numpy	65.63%	26.87%	23.53%
Opengl	33.73%	39.36%	20.80%
React	75.56%	20.00%	7.93%
JDBC	26.36%	61.82%	8.11%
Average	55.13%	33.64%	19.21%

The remainder of the paper is organized as follows. Section 2 reports our empirical studies of the characteristics of API-names, API-mentions and discussion contexts. Section 3 defines the problem of API extraction. Section 4 and Section 5 describe our neural architecture for API extraction and the system implementation respectively. Section 6 reports our experiment results and findings. Section 7 reviews the related work. Section 8 concludes our work and discuss the future work.

2 Empirical Studies of API-Naming and API-Mention Characteristics

In this work, we aim to develop machine learning based API extraction method that is not only effective but also easy-to-deploy across programming languages and libraries. To understand the challenges in achieving this objective and the potential solution space, we conduct empirical studies of the characteristics of API names, API mentions in informal texts, and discussion contexts in which APIs are mentioned.

We study six libraries: three Python libraries: *Matplotlib* (data visualization), *Pandas* (machine learning), *Numpy* (numeric computation), one C library *OpenGL* (computer graphics), one JavaScript library *React* (graphical user interface), and one Java library *JDBC* (database access). These libraries come from the four popular programming languages, and they support very diverse functionalities for computer programming.

First, we crawl API declarations of these libraries from their official websites. When different APIs have a same simple name but different arguments in a same library, we treat such APIs as the same. We examine each API name to determine if the simple name of an API is a common word (e.g, apply, series, draw) that can be found in a general English dictionary. We find that different libraries have different percentages of APIs with common-word simple names: *OpenGL* (6%), *Matplotlib* (16%), *Numpy* (41%), *JDBC* (43%), *Pandas* (55%), *React* (66%). When these APIs are mentioned by their common-word simple names, neither character- nor word-level features can help to distinguish such polysemous API mentions from common words. We must resort to discussion contexts of API mentions.

Second, by checking post tags, we randomly sample 200 Stack Overflow posts for each of the six libraries. We manually label API mentions in these posts. We examine three characteristics of API mentions: whether API mentions contain explicit orthographic features (package or module names, parentheses, and/or dot), whether API mentions are long tokens (> 10 characters), and whether the context windows (preceding and succeeding 5 words) around the API mentions contain common verbs and nouns (use, call, function, method). Table 2 shows our analysis results. On average 55.13% API mentions contain explicit orthographic, and 33.64% API mentions are long tokens. Character-level features would be useful for recognizing these API mentions. However, for the significant amount of API mentions that do not have such explicit character-level features, we need to resort to word- and/or sentence-content features, for example, the words (e.g., use, call, function, method) that often appear in the context window of an API mention, to recognize API mentions.

Furthermore, we can observe that API-mention characteristics are not tightly coupled with a particular programming language or library. Instead, all six libraries exhibit certain degree of the three API-mention characteristics. But specific degrees vary across libraries. Fig. 2 visualizes the top 50 frequently-used words in the discussions of the six libraries. We can observe that discussions of different libraries share common words, but at the same time use library-specific words (e.g., dataframe for *Pandas*, matrix for *Numpy*, figure for *Matplotlib*, query for *JDBC*, render for *OpenGL*, event for *React*). The commonalities of API mention characteristics across libraries indicate the feasibility of transfer learning. For example, orthographic, word and/or sentence-context features learned from a source library could be applicable to a target library. However, due to the variations of API-name, API-mention and discussion-context characteristics across libraries, directly applying the source-library trained model to the target library may suffer from performance degradation, unless the source and target libraries have very similar characteristics. Therefore, fine-tuning the source-library trained model with a small amount of target library text would be necessary.

3 Problem Definition

In this work, we takes as input *informal software text* (e.g., Stack Overflow posts) that discusses the usage and issues of a particular library. We assume that the software text of multiple programming languages and libraries need to be processed. Given a paragraph of informal software text, our task is to recognize all API mentions (if any) in the paragraph, as illustrated in the example in Fig. 1. API mentions refer to tokens in the paragraph that represent public modules, classes, methods or functions of a particular library. To preserving the integrity of code-like tokens, an API mention is defined as a single token rather than a span of tokens when the given text is tokenized properly.

As our input is informal software text, APIs may not be consistently mentioned in their formal full names. Instead, APIs may be mentioned in abbreviations or synonyms, such as *pandas's dataframe* for *panads.DataFrame*, *df.apply* for

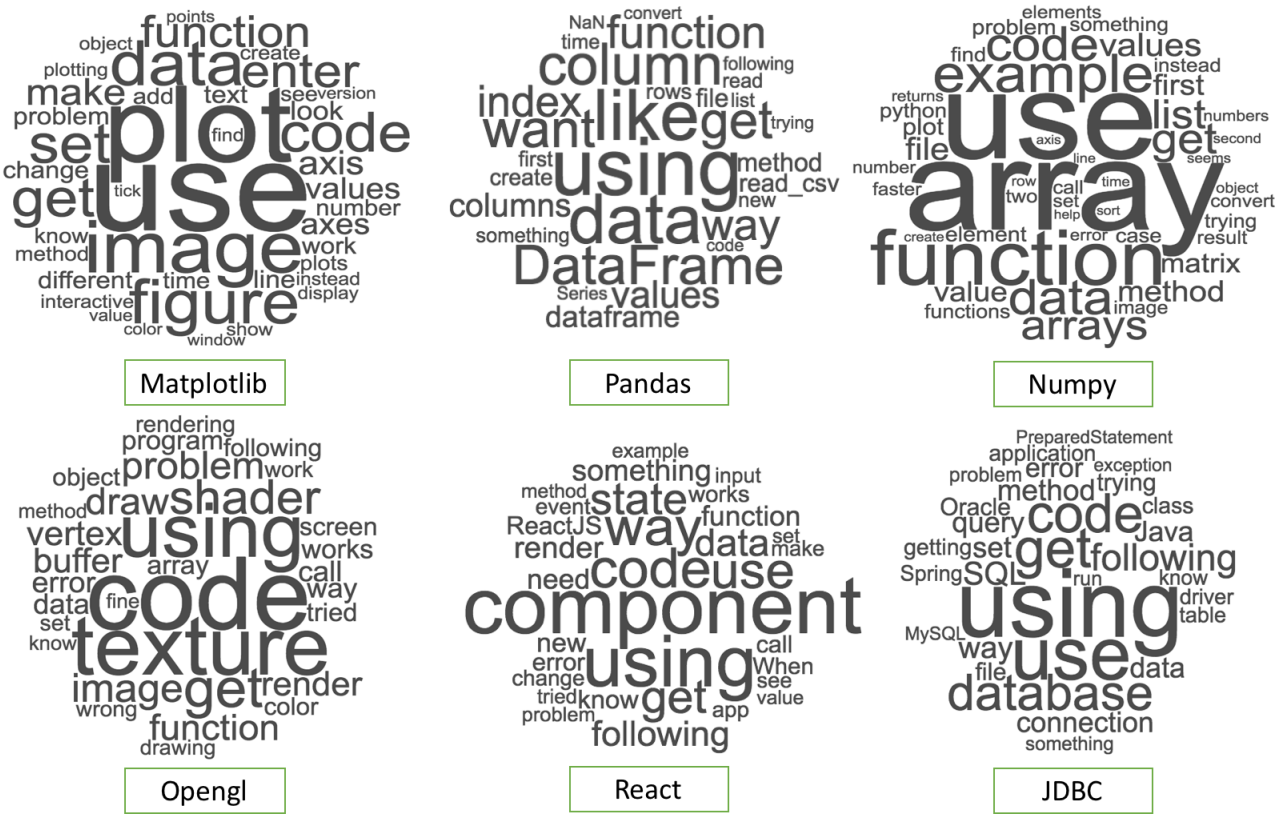

Fig. 2: Word Clouds of the Top 50 Frequent Words in the Discussions of the Six Libraries

pandas.DataFrame.apply(). APIs may also be mentioned in their simple names, such as *apply*, *series*, *dataframe*, that can also be common English words or computing terms in the text. Furthermore, we do not assume that API mentions will be consistently annotated with special tags. Therefore, our approach takes plain text as input.

A related task to our work is API linking. API extraction methods classify whether a token in text is an API mention or not, while API linking methods link API mentions in text to API entities in a knowledge base [15]. That is, API extraction is the prerequsite for API linking. This work deals with only API extraction.

4 Our Neural Architecture

We formulate the task of API extraction in informal software texts as a sequence labeling problem, and present a neural architecture that labels each token in an input text as API or non-API. As shown in Fig. 3, our neural architecture is composed of four main components: 1) a character-level Convolutional Neural Network (CNN) for extracting character-level features of a token (Section 4.1), 2) an unsupervised word embedding model for learning word semantics of a token (Section 4.2), 3) a Bidirectional Long Short-Term Memory network (Bi-LSTM) for extracting sentence-context features (Section 4.3), and 4) a softmax classifier for predicting the API or non-API label of a token (Section 4.4). Our neural model can be trained end-to-end with pairs of input texts and their corresponding API/non-API label sequences. A model trained with one library's text can be transferred to another library's text by fine-tuning source-library-trained components with the target library's text. (Section 4.5).

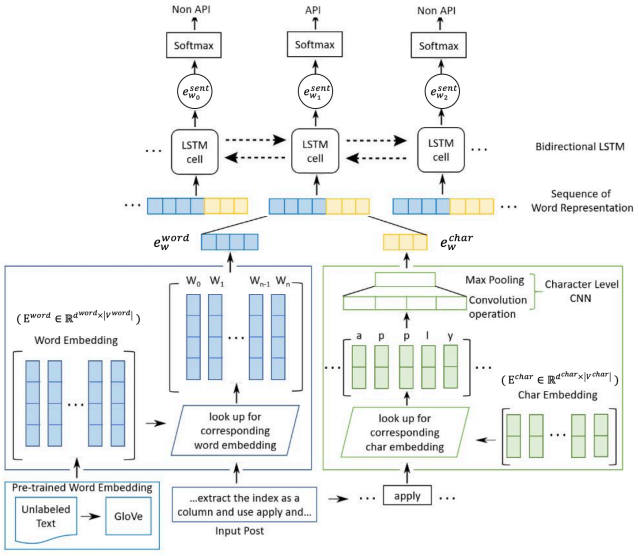

Fig. 3: Our Neural Architecture for API Extraction

4.1 Extracting Char-Level Features by CNN

API mentions often have morphological features that distinguish them from normal English words. Such morphological features may appear in the beginning of a token (e.g., the first letter capitalization *System*), in the middle (e.g., the hyphen in *read_csv*, the dot in *pandas.series*, the left

abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRS
TUVWXYZ0123456789,.;!?:'+-*=/\()[]{}|_@#$%&~'^~

Fig. 4: Our Character Vocabulary

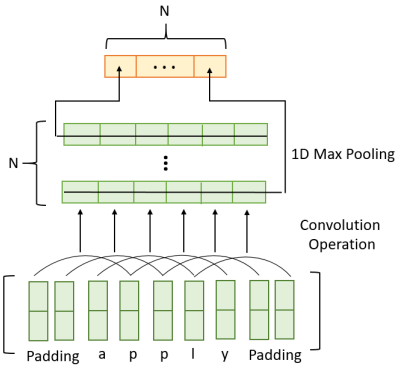

Fig. 5: Character-Level CNN

parenthesis and comma in *print(a,32)*), or at the end (e.g., the right parenthesis in *apply()*). Morphological features may also appear in combination, such as camelcase writing like *AbstractAction*, a pair of parentheses like *plot()*. The long length of some tokens like *createdataset* is one important morphological feature as well. Due to the lack of universal naming convention across libraries and the wide presence of informal writing forms, informative morphological features of API mentions often vary from one library's text to another.

Robust methods to extract morphological features from tokens must take into account all characters of the token, and determine which features are more important for a particular library's APIs [16]. To that end, we use a character-level CNN [17], which extracts local features in N-gram characters of the token using a convolution operation and then combines them using a max-pooling operation to create a fixed-sized character-level embedding of the token [18], [19].

Let V^{char} be the vocabulary of characters for the software texts from which we want to extract API mentions. In this work, V^{char} for all of our models consists of 92 characters, including 26 English letters (both upper and lower case), 10 digits, 30 other characters (e..g, ':','[','?'), as listed in Fig. 4. Note that V^{char} can be easily extended for different datasets. Let $E^{char} \in \mathbb{R}^{d^{char} \times |V^{char}|}$ be the character embedding matrix where d^{char} is the dimension of character embeddings and $|V^{char}|$ is the vocabulary size (92 in this work). As illustrated in Fig. 3, E^{char} can be regarded as a dictionary of character embeddings in which a column d^{char}-dimensional vector corresponds to a particular character. The character embeddings are initialized as one-hot vectors and then learned during the training of character-level CNN. Given a character $c \in V^{char}$, its embedding e^c can be retrieved by the matrix-vector product $e^c = E^{char} v^c$ where v^c is a one-hot vector of size $|V^{char}|$ which has value 1 at index c and zero in all other positions.

Fig. 5 presents the architecture of our character-level CNN. Given a token w in the input text, let's assume w is composed of M characters $\{c_1, c_2, ..., c_M\}$. We first obtain a sequence of character embeddings $\{e^{c_1}, e^{c_2}, ..., e^{c_M}\}$ by looking up the character embeddings matrix E^{char}. This sequence of character embeddings (zero-padding at the beginning and the end of the sequence) is the input to our character-level CNN. In our application of CNN, because each character is represented as a d^{char}-dimensional vector, we use convolution filters with widths equal to the dimensionality of the character embeddings (i.e., d^{char}). Then we can vary the height h (or window size) of the filter, i.e., the number of adjacent characters considered jointly in the convolution operation.

Let z_m be the concatenation of the character embeddings of c_m $(1 \leq m \leq M)$, the $(h-1)/2$ left neighbors of c_m, and the $(h-1)/2$ right neighbors of c_m. A convolution operation involves a filter $W \in \mathbb{R}^{hd^{char}}$ (a matrix of $h \times d^{char}$ dimensions) and a bias term $b \in \mathbb{R}^h$, which is applied repeatedly to each character window of size h in the input sequence $\{c_1, c_2, ..., c_M\}$:

$$o_m = ReLU(W^T \cdot z_m + b)$$

where $ReLU(x) = max(0, x)$ is the non-linear activation function. The convolution operations produce a M-dimensional feature map for a particular filter. A 1D-max pooling operation is then applied to the feature map to extract a scalar (i.e., a feature vector of length 1) with the maximum value in the M dimensions of the feature map.

The convolution operation extracts local features within each character window of the given token, and using the max over all character windows of the token, we extract a global feature for the token. We can apply N filters to extract different features from the same character window. The output of the character-level CNN is an N-dimensional feature vector representing the character-level embedding of the given token. We denote this embedding as e^{char}_w for the token w.

In our character-level CNN, the matrices E^{char} and W, and the vector b are parameters to be learned. The dimensionality of the character embedding d^{char}, the number of filters N, and the window size of the filters h are hyper-parameters to be chosen by the user (see Section 5.2 for model configuration).

4.2 Learning Word Semantics by Word Embedding

In informal software texts, the same API is often mentioned in many non-standard abbreviations and synonyms [5]. For example, the *Pandas* library is often written as *pd*, and its module *DataFrame* is often written as *df*. Furthermore, there is lack of consistent use of verb, noun and preposition in the discussions [2]. For example, in the sentences "I have decided to use apply ...", "if you run apply on a series ...", and "I tested with apply ...", users refer to a *Pandas*'s method *apply()*, but their descriptions vary greatly.

Such variations result in out-of-vocabulary (OOV) issue for a machine learning model [20], i.e., variations that have not been seen in the training data. For the easy deployment of a machine learning model for API extraction, it is impractical to address the OOV issue by manually labeling a huge amount of data and developing a comprehensive gazetteer of API names and their common variations [2]. However, without the knowledge about variations of semantically

similar words, the trained model will be restricted to the examples that it sees in the training data.

To address this dilemma, we propose to exploit unsupervised word-embedding models [21], [22], [23], [24] to learn distributed word representations from a large amount of unlabeled text discussing a particular library. Many studies [25], [26], [27] have shown that distributed word representations can capture rich semantics of words such that semantically similar words will have similar word embeddings.

Let V^{word} be the vocabulary of words for the corpus of software texts to be processed. As illustrated in Fig. 3, word-level embeddings are encoded by column vectors in a word embedding matrix $E^{word} \in \mathbb{R}^{d^{word} \times |V^{word}|}$ where d^{word} is the dimension of word embeddings and $|V^{word}|$ is the vocabulary size. Each column $E_i^{word} \in \mathbb{R}^{d^{word}}$ corresponds to the word-level embedding of the i-th word in the vocabulary V^{word}. We can obtain a token w's word-level embedding e_w^{word} by looking up E^{word} with the word w, i.e., $e_w^{word} = E^{word} v^w$ where v^w is a one-hot vector of size $|V^{word}|$ which has value 1 at index w and zero in all other positions. The matrix E^{word} is to be learned using unsupervised word embedding models (e.g., GloVe [28]), and d^{word} is a hyper-parameter to be chosen by the user (see Section 5.2 for model configuration).

In this work, we adopt the Global Vectors for Word Representation (GloVe) method [28] to learn the matrix E^{word}. GloVe is an unsupervised algorithm for learning word representations based on the statistics of word co-occurrences in an unlabeled text corpus. It calculates the word embeddings based on a word co-occurrence matrix X. Each row in the word co-occurrence matrix corresponds to a word, and each column corresponds to a context. X_{ij} is the frequency of word i co-occurring with word j, and $X_i = \sum X_{ik}$ $(1 \leq k \leq |V^{word}|)$ is the total number of occurrences of word i in the corpus. The probability of word j that occurs in the context of word i is $P_{ij} = P(j|i) = X_{ij}/X_i$. We have $log(P_{ij}) = log(X_{ij}) - log(X_i)$.

GloVe defines $log(P_{ij}) = e_{w_i}^T e_{w_j}$ where e_{w_i} and e_{w_j} are the word embeddings to be learned for the word w_i and w_j. This gives the constraint for each word pair as $log(X_{ij}) = e_{w_i}^T e_{w_j} + b_i + b_j$ where b is the bias term for e_w. The cost function for minimizing the loss of word embeddings is defined as:

$$\sum_{i,j=1}^{V^{word}} f(X_{ij})(e_{w_i}^T e_{w_j} + b_i + b_j - log(X_{ij}))$$

where $f(X_{ij})$ is a weighting function. That is, GloVe learns word embeddings by a weighted least square regression model.

4.3 Extracting Sentence-Context Features by Bi-LSTM

In informal software texts, many API mentions cannot be reliably recognized by simply examining a token's character-level features and word semantics. This is because many APIs are named using common English words (e.g., *series*, *apply*, *plot*) or common computing terms (e.g., *dataframe*, *sigmoid*, *histgram*, *argmax*, *zip*, *list*). When such APIs are mentioned in their simple name, this results in a common-word polesemy issue for API extraction [2]. In such situations, we have to disambiguate the API sense of a common word from the normal sense of the word.

To that end, we have to look into the sentence context in which an API is mentioned. For example, by looking into the sentence context of the two sentences in Fig. 1, we can determine that the "apply" in the first sentence is an API mention, while the "apply" in the second sentence is not an API mention. Note that both the preceding and succeeding context of the token "apply" are useful for disambiguating the API or the normal sense of the token.

We use Recurrent Neural Network (RNN) to extract sentence context features for disambiguating the API or the normal sense of the word [29]. RNN is a class of neural networks where connections between units form directed cycles and it is widely used in software engineering domain [30], [31], [32], [33]. Due to this nature, it is especially useful for tasks involving sequential inputs [29] like sentences. In our task, we adopt a Bidirectional RNN (Bi-RNN) architecture [34], [35], which is composed of two LSTMs, one takes input from the beginning of the text forward till a particular token, while the other takes input from the end of the text backward till that token.

The input to an RNN is a sequence of vectors. In our task, we obtain the input vector of a token in the input text by concatenating the character-level embedding of the token w and the word-level embedding of the token w, i.e., $e_w^{char} \oplus e_w^{word}$. An RNN recursively maps an input vector x_t and a hidden state h_{t-1} to a new hidden state h_t: $h_t = f(h_{t-1}, x_t)$ where f is a non-linear activation function (e.g., an LSTM unit used in this work). A hidden state is a vector $e_{sent} \in \mathbb{R}^{d^{sent}}$ summarizing the sentence-level features till the input x_t, where d^{sent} is the dimension of the hidden state vector to be chosen by the user. We denote e_{sent}^f and e_{sent}^b as the hidden states computed by the forward LSTM and the backward LSTM after reading the end of the preceding and the succeeding sentence context of a token, respectively. e_{sent}^f and e_{sent}^b are concatenated into one vector e_w^{sent} as the Bi-LSTM output for the corresponding token w in the input text.

As an input text (e.g., a Stack Overflow post) can be a long text, modeling long-range dependencies in the text is crucial for our task. For example, a mention of a library name at the beginning of a post could be important for detecting a mention of a method of this library later in the post. Therefore, we adopt the LSTM unit [36], [37] in our RNN. The LSTM is designed to cope with the gradient vanishing problem in RNN. An LSTM unit consists of a memory cell and three gates, namely the input, output and forget gates. Conceptually, the memory cell stores the past contexts, and the input and output gates allow the cell to store contexts for a long period of time. Meanwhile, some contexts can be cleared by the forget gate. Memory cell and the three gates have weights and bias terms to be learned during model training. Bi-LSTM can extract the context feature. For instance, in the Pandas library sentence "This can be accomplished quite simply with the DataFrame method apply", based on the context information from the word *method*, the Bi-LSTM can help our model classify the word *apply* as API mention, and the learning can be

transferred across different languages and libraries with transfer learning.

4.4 API Labeling by Softmax Classification

Given the Bi-LSTM's output vector e_w^{sent} for a token w in the input text, we train a binary softmax classifier to predict whether the token is an API mention or a normal English word. That is, we have two classes in this work, i.e., API or non-API. Softmax predicts the probability for the j-th class given a token's vector e_w^{sent} by:

$$P(j|e_w^{sent}) = \frac{exp(e_w^{sent}W_j^T)}{\sum_{k=1}^{2} exp(e_w^{sent}W_k^T)}$$

where the vectors W_k (k=1 or 2) are parameters to be learned.

4.5 Library Adaptation by Transfer Learning

Transfer learning [12], [38] is a machine learning method that stores knowledge gained while solving one problem and applying it to a different but related problem. It helps to reduce the amount of training data required for the target problem. When transfer learning is used in deep learning, it has been shown to be beneficial for narrowing down the scope of possible models on a task by using the weights of a trained model on a different but related task [14], and the shared parameters transfer learning can help model adapt the shared knowledge in similar context [39].

API extraction tasks for different libraries can be regarded as a set of different but related tasks. Therefore, we use transfer learning to adapt a neural model trained with one library's text (source-library-trained model) for another library's text (target library). Our neural architecture is composed of four main components: character-level CNN, word embeddings, sentence Bi-LSTM, and softmax classifier. We can transfer all or some source-library-trained model components to a target library model. Without transfer learning, the parameters of a target-library model component will be randomly initialized and then learned using the target-library training data, i.e., trained from scratch. With transfer learning, we use the parameters of source-library-trained model components to initialize the parameters of the corresponding target-library model components. After transferring the model parameters, we can either freeze the transferred parameters or fine-tune the transferred parameters using the target-library training data.

5 System Implementation

This section describes the current implementation [2] of our neural architecture for API extraction.

5.1 Preprocessing and Tokenizing Input Text

Our current implementation takes as input the content of a Stack Overflow post. As we want to recognize API mentions within natural language sentences, we remove stand-alone code snippets in <pre><code> tag. We then remove all HTML tags in the post content to obtain the plain text input (see Section 3 for the justification of taking plain text as input). We develop a sentence parser to split the preprocessed post text into sentences by punctuation. Following [2], we develop a software-specific tokenizer to tokenize the sentences. This tokenizer preserves the integrity of code-like tokens. For example, it treats *matplotlib.pyplot.imshow()* as a single token, instead of a sequence of 7 tokens, i.e., "matplotlib" "." "pyplot" "." "imshow" "(" ")" produced by general English tokenizers.

2. https://github.com/JOJO201/API_Extraction

5.2 Model Configuration

We now describe the hyper-parameter settings used in our current implementation. These hyper-parameters are tuned using the validation data during model training (see Section 6.1.2 for the description of our dataset). We find that the neural model has very similar performance across six libraries dataset with the same hyper-parameters. Therefore, we keep the same hyper-parameters for the six libraries dataset, which can also avoid the difficulty in scaling different hyper-parameters.

5.2.1 Character-level CNN

We set the filter window size $h = 3$. That is, the convolution operation extracts local features from 3 adjacent characters at a time. The size of our current character vocabulary V^{char} is 92. Thus, we set the dimensionality of the character embedding d^{char} at 92. We initialize the character embedding with one-hot vector (1 at one character index and zero in all other dimensions). The character embeddings will be updated through back propagation during the training of character-level CNN. We experiment 5 different N (the number of filters): 20, 40, 60, 80, 100. With $N = 40$, the CNN has an acceptable performance on the validation data. With $N = 60$ and above, the CNN has almost the same performance as $N = 40$, but it takes more training epochs to research the acceptable performance. Therefore, we use $N = 40$. That is, the character-level embedding of a token has the dimension 40.

5.2.2 Pre-trained word embeddings

Our experiments involve six Python libraries: *Pandas*, *Matplotlib*, *NumPy*, *OpenGL*, *React* and *JDBC*. We collect all questions tagged with these six libraries and all answers to these questions in the Stack Overflow Data Dump released on March 18, 2018. We obtain a text corpus of 380971 posts. We use the same preprocessing and tokenization steps as described in Section 5.1 to preprocess and tokenize the content of these posts. Then, we use the GloVe [28] to learn word embeddings from this text corpus. We set the word vocabulary size $|V^{word}|$ at 40000. The training epoch is set at 100 to ensure the sufficient training of word embeddings. We experiment four different dimensions of word embeddings d^{word}: 50, 100, 200, 400. We use $d^{word} = 200$ in our current implementation as it produces a good balance of the training efficiency and the quality of word embeddings for API extraction on the validation data. We also experiment pre-trained word embeddings with all Stack Overflow posts, which does not significantly affect the API extraction performance but requires much longer time for text preprocessing and word embeddings learning.

5.2.3 Sentence-context Bi-LSTM

For the RNN, we use 50 hidden LTSM units to store the hidden states. The dimension of the hidden state vector is 50. Therefore, the dimension of the output vector e_w^{sent} is 100 (concatenating forward and backward hidden states). In order to mitigate overfitting [40], a dropout layer is added on the output of BLSTM, with the dropout rate 0.5.

5.3 Model Training

To train our neural model, we use input text and its corresponding sequence of API/non-API labels (see Section 6.1.2 for our data labeling process). The optimizer used is Adam [41], which performs well for training RNN including Bi-LSTM [35]. The training epoch is set at 40 times. The best performance model on the validation set is saved for testing. This model's parameters are also saved for the transfer learning experiments.

6 Experiments

We conduct a series of experiments to answer the following four research questions:

- **RQ1**: How well can our neural architecture learn multi-level features from the input texts? Can the learned features support high-quality API extraction, compared with existing machine learning based API extraction methods?
- **RQ2**: What is the impact of the three feature extractors (character-level CNN, word embeddings, and sentence-context Bi-LSTM) on the API extraction performance?
- **RQ3**: How effective is transfer learning for adapting API extraction models across libraries of the same language with different amount of target-library training data?
- **RQ4**: How effective is transfer learning for adapting API extraction models across libraries of different languages with different amount of target-library training data?

6.1 Experiments Setup

This section describes the libraries used in our experiments, how we prepare training and testing data, and the evaluation metrics of API extraction performance.

6.1.1 Studied libraries

Our experiments involve three Python libraries (*Pandas*, *NumPy* and *Matplotlib*), one Java library (*JDBC*), one JavaScript library (*React*), and one C library (*OpenGL*). As reported in Section 2, these six libraries support very diverse functionalities for computer programming, and have distinct API-naming and API mention characteristics. Using these libraries, we can evaluate the effectiveness of our neural architecture for API extraction in very diverse data settings. We can also gain insights into the effectiveness of transfer learning for API extraction and the transferability of our neural architecture in different settings.

6.1.2 Dataset

We collect Stack Overflow posts (questions and their answers) tagged with *pandas*, *numpy*, *matplotlib*, *opengl*, *react* or *jdbc* as our experimental data. We use Stack Overflow Data Dump released on March 18, 2018. In this data dump, 380971 posts are tagged with one of the six studied libraries. Our data collection process follows four criteria. First, the number of posts selected and the number of API mentions in these posts should be at the same order of magnitude. Second, API mentions should exhibit the variations of API writing forms commonly seen on Stack Overflow. Third, the selection should avoid repeatedly selecting frequently mentioned APIs. Fourth, the same post should not appear in the dataset for different libraries (one post may be tagged with two or more studied-library tags). The first criterion ensures the fairness of comparison across libraries, the second and third criteria ensure the representativeness of API mentions in the dataset, and the fourth criterion ensures that there is no repeated data in different datasets.

TABLE 3: Basic Statistics of Our Dataset

Library	Posts	Sentences	API mentions	Tokens
Matplotlib	600	4920	1481	47317
Numpy	600	2786	1552	39321
Pandas	600	3522	1576	42267
Opengl	600	3486	1674	70757
JDBC	600	4205	1184	50861
React	600	3110	1262	42282
Total	3600	22029	8729	292805

We finally include 3600 posts (600 for each library) in our dataset and each post has at least one API mention.[3] Table 3 summarizes the basic statistics of our dataset. These posts have in total 22029 sentences, 292805 token occurrences, and 41486 unique tokens after data preprocessing and tokenization. We manually label the API mentions in the selected posts. 6421 sentences (34.07%) contains at least one API mention. Our dataset has in total 8729 API mentions. The selected *Matplotlib*, *NumPy*, *Pandas*, *OpenGL*, *JDBC* and *React* posts have 1481, 1552, 1576, 1674, 1184 and 1262 API mentions, which refer to 553, 694, 293, 201, 282 and 83 unique APIs of the six libraries, respectively. In addition, we randomly sample 100 labelled Stack Overflow posts for each library dataset. Then, we examine each API mention to determine whether it is a simple name or not, and we find that among the 600 posts, there are 1634 API mentions of which 694 (42.47%) are simple API names. Our dataset not only contains a large number of API mentions in diverse writing forms, but also contains rich discussion context around API mentions. This makes it suitable for the training of our neural architecture, the testing of its performance, and the study of model transferability.

We randomly split the 600 posts of each library into three subsets by 6:2:2 ratio: 60% as training data, 20% as validation data for tuning model hyper-parameters, and 20% as testing data to answer our research questions. Since cross validation will cost a great deal of time and our training dataset is unbiased (proved in Section 6.4), cross validation is not used. Our experiment results also show that the training dataset is enough to train a high-performance neural network. Besides, the performance of the model in target library can be improved by adding more training dataset from other libraries. And if more training dataset is added, there is no need to label a new validate dataset for the target library to re-tune the hyper-parameters, because we find that the

3. https://drive.google.com/drive/folders/1f7ejNVUsew9l9uPCj4Xv5gMzNbqttpoa?usp=sharing

TABLE 4: Comparison of CRF Baselines [2] and Our Neural Model for API Extraction

	Basic CRF			Full CRF			Our method		
Library	Prec	Recall	F1	Prec	Recall	F1	Prec	Recall	F1
Matplotlib	67.35	47.84	56.62	89.62	61.31	72.82	81.5	83.92	82.7
NumPy	75.83	39.91	52.29	89.21	49.31	63.51	78.24	62.91	69.74
Pandas	62.06	70.49	66.01	97.11	71.21	82.16	82.73	85.3	82.80
Opengl	43.91	91.87	59.42	94.57	70.62	80.85	85.83	83.74	84.77
JDBC	15.83	82.61	26.58	87.32	51.40	64.71	84.69	55.31	66.92
React	16.74	88.58	28.16	97.42	70.11	81.53	87.95	78.17	82.77

neural model has very similar performance with the same hyper-parameters under different dataset settings.

6.1.3 Evaluation metrics

We use precision, recall, and F1-score to evaluate the performance of an API extraction method. Precision measures what percentage the recognized API mentions by a method are correct. Recall measures what percentage the API mentions in the testing dataset are recognized correctly by a method. F1-score is the harmonic mean of precision and recall, i.e., $2 * ((precision * recall)/(precision + recall))$.

6.2 Performance of Our Neural Model (RQ1)

Motivation: Recently, linear-chain CRF has been used to solve the API extraction problem in informal text [2]. They show that machine-learning based API extraction with that of several outperforms several commonly-used rule-based methods [2], [42], [43]. The approach in [2] relies on human-defined orthographic features, two different unsupervised language models (class-based Brown clustering and neural-network based word embedding) and API gazetteer features (API inventory). In contrast, our neural architecture uses neural networks to automatically learn character-, word- and sentence-level features from the input texts. The first RQ is to confirm the effectiveness of these neural-network feature extractors for API extraction tasks by comparing the overall performance of our neural architecture with the performance of the linear-chain CRF with human-defined features for API extraction.

Approach: We use the implementation of the CRF model in [2] to build the two CRF baselines. The basic CRF baseline uses only the orthographic features developed in [2]. The full CRF baseline uses all orthographic, word-clusters and API gazetteer features developed in [2]. The basic CRF baseline is easy to deploy because it uses only the orthographic features in the input texts, but not any advanced word-clusters and API gazetteer features. And the self-training process is not used in these two baselines, since we have sufficient training data. In contrast, the full CRF baseline uses advanced features so that it is not as easy-to-deploy as the basic CRF. But the full CRF has much better performance than the basic CRF as reported in [2]. Comparing our model with these two baselines, we can understand whether our model can achieve a good tradeoff between easy-to-deploy and high performance. Note that as Ye et al. [2] already demonstrates the superior performance of machine-learning based API extraction methods over rule-based methods, we do not consider rule-based baselines for API extraction in this study.

Results: From Table 4, we can see that:

- *Although linear CRF with full features has close performance to our model, linear CRF with only orthographic features performs poorly in distinguishing API tokens from non-API tokens*. The best F1-score of the basic CRF is only 0.66 for *Pandas*. The F1-score of the basic CRF for *Matplotlib*, *Numpy* and *OpenGL* is 0.52-0.59. For *JDBC* and *React*, the F1-score of the basic CRF is below 0.3. Our results are consistent with the results in [2] when advanced word-clusters and API-gazetteer features are ablated. Compared with the basic CRF, the full CRF performs much better. For *Pandas*, *JDBC* and *React*, the full CRF has very close performance to our model. But for *Matplotlib*, *Numpy* and *OpenGL*, our model still outperforms the full CRF by at least four points in F1-score.
- *Multi-level feature embedding by our neural architecture is effective in distinguishing API tokens from non-API tokens in the resulting embedding space*. All evaluation metrics of our neural architecture are significantly higher than those of the basic CRF. Although our neural architecture performs relatively worse on *NumPy* and *JDBC*, its F1-score is still much higher than the F1-score of the basic CRF on *NumPy* and *JDBC*. We examine false positives (non-API token recognized as API) and false negatives (API token recognized as non-API) by our neural architecture on *NumPy* and *JDBC*. We find that many false positives and false negatives involve API mentions composed of complex strings, for example, array expressions for *Numpy* or SQL statements for *JDBC*. It seems that neural networks learn some features from such complex strings that may confuse the model and cause the failure to tell apart API tokens from non-API ones. Furthermore, by analysing the classification results for different API mentions, we find that our model has good generalizability on different API functions.

With linear CRF for API extraction, we cannot achieve a good tradeoff between easy-to-deploy and high performance. In contrast, our neural architecture can extract effective character-, word- and sentence-level features from the input texts and the extracted features alone can support high-quality API extraction, without using any hand-crafted advanced features such as word clusters, API gazetteers.

6.3 Impact of Feature Extractors (RQ2)

Motivation: Although our neural architecture achieves very good performance as a whole, we would like to further investigate how much different feature extractors (character-level CNN, word embeddings, and sentence-context Bi-LSTM) contribute to this good performance, and how different features affect precision and recall of API extraction.

TABLE 5: The Results of Feature Ablation Experiments

	Ablating char CNN			Ablating Word Embeddings			Ablating Bi-LSTM			All features		
	Prec	Recall	F1-score	Prec	Recall	F1-score	Prec	Recall	F1-score	Prec	Recall	F1-score
Matplotlib	**75.70**	72.32	73.98	81.75	**63.99**	**71.79**	82.59	69.34	75.44	81.50	83.92	82.70
Numpy	81.00	60.40	69.21	79.31	58.47	67.31	80.62	**56.67**	**66.44**	78.24	62.91	69.74
Pandas	**80.50**	83.28	81.82	81.77	**68.01**	**74.25**	83.22	75.22	79.65	82.73	85.30	82.80
Opengl	**77.58**	85.37	81.29	83.07	**68.03**	**75.04**	98.52	72.08	83.05	85.83	83.74	84.77
JDBC	68.65	61.25	**64.47**	**64.22**	65.62	64.91	99.28	**43.12**	66.13	84.69	55.31	66.92
React	**69.90**	85.71	**77.01**	84.37	75.00	79.41	98.79	**65.08**	78.47	87.95	78.17	82.77

Approach: We ablate one kind of feature at a time from our neural architecture. That is, we obtain a model without character-level CNN, one without word embeddings, and one without sentence-context Bi-LSTM. We compare the performance of these three models with that of the model with all three feature extractors.

Results: In Table 5, we highlight in bold the largest drop in each metric for a library when ablating a feature, compared with that metric with all features. We underline the increase in each metric for a library, when ablating a feature, compared with that metric with all features. We can see that:

- *The performance of our neural architecture is contributed by the combined action of all its features*. Ablating any of the features, the F1-score degrades. Ablating word embeddings or Bi-LSTM causes the largest drop in F1-score for four libraries, while ablating char-CNN causes the largest drop in F1-score for two libraries. Feature ablation has higher impact on some libraries than others. For example, ablating char-CNN, word embeddings and Bi-LSTM all cause significant drop in F1-score for *Matplotlib*, and ablating word embeddings causes significant drop in F1-score for *Pandas* and *React*. In contrast, ablating a feature causes relative minor drop in F1-score for *Numpy*, *JDBC* and *React*. This indicates different levels of features learned by our neural architecture can be more distinct for some libraries, but more complementary for others. When different features are more distinct from one another, achieving good API extraction performance relies more on all the features.
- *Different features play different roles in distinguishing API tokens from non-API tokens*. Ablating char-CNN causes the drop in precision for five libraries, except *Numpy*. The precision degrades rather significantly for *Matplotlib* (7.1%), *OpenGL* (9.6%), *JDBC* (18.9%) and *React* (20.5%). In contrast, ablating char-CNN causes significant drop in recall only for *Matplotlib* (13.8%). For *JDBC* and *React*, ablating char-CNN even causes significant increase in recall (10.7% and 9.6% respectively). Different from the impact of ablating char-CNN, ablating word embeddings and Bi-LSTM usually causes the largest drop in recall, ranging from 9.9% drop for *Numpy* to 23.7% drop for *Matplotlib*. Ablating word embeddings causes significant drop in precision only for *JDBC*, and it causes small changes in precision for the other five libraries (three with small drop and two with small increase). Ablating Bi-LSTM causes the increase in precision for all six libraries. Especially for *OpenGL*, *JDBC* and *React*, when ablating Bi-LSTM, the model has almost perfect precision (around 99%), but this comes with a significant drop in recall (13.9% for *OpenGL*, 19.3% for *JDBC* and 16.7% *React*).

All feature extractors are important for high-quality API extraction. Char-CNN is especially useful for filtering out non-API tokens, while word embeddings and Bi-LSTM are especially useful for recalling API tokens.

6.4 Effectiveness of Within-Language Transfer Learning (RQ3)

Motivation: As described in Section 6.1.1, we intentionally choose three different-functionality Python libraries: *Pandas*, *Matplotlib*, *NumPy*. *Pandas* and *NumPy* are functionally similar, while the other two pairs are more distant. The three libraries also have different API-naming and API-mention characteristics (see Section 2). We want to investigate the effectiveness of transfer learning for API extraction across different pairs of libraries and with different amount of target-library training data.

Approach: We use one library as source library and one of the other two libraries as target library. We have six transfer-learning pairs for the three libraries. We denote them as *source-library-name* $\rightarrow$ *target-library-name*, such as *Pandas* $\rightarrow$ *NumPy* which represents transferring *Pandas*-trained model to *NumPy* text. Two of these six pairs (*Pandas* $\rightarrow$ *NumPy* and *NumPy* $\rightarrow$ *Pandas*) are similar-libraries transfer (in terms of library functionalities and API-naming/mention characteristics), while the other four pairs involving *Matplotlib* are relatively more-distant-libraries transfer.

TABLE 6: NumPy (NP) or Pandas (PD) $\rightarrow$ Matplotlib (MPL)

	NP$\rightarrow$MPL			PD$\rightarrow$MPL			MPL		
	Prec	Recall	F1	Prec	Recall	F1	Prec	Recall	F1
1/1	82.64	89.29	85.84	81.02	80.06	80.58	87.67	78.27	82.70
1/2	81.84	84.52	83.16	71.61	83.33	76.96	81.38	70.24	75.40
1/4	71.83	75.89	73.81	67.88	77.98	72.65	81.22	55.36	65.84
1/8	70.56	75.60	72.98	69.66	73.81	71.71	75.00	53.57	62.50
1/16	73.56	72.02	72.78	66.48	72.02	69.16	80.70	27.38	40.89
1/32	72.56	78.83	73.69	71.47	69.35	70.47	97.50	11.60	20.74
DU	72.54	66.07	69.16	76.99	54.76	64.00			

TABLE 7: Matplotlib (MPL) or Pandas (PD) $\rightarrow$ NumPy (NP)

	MPL$\rightarrow$NP			PD$\rightarrow$NP			NP		
	Prec	Recall	F1	Prec	Recall	F1	Prec	Recall	F1
1/1	86.85	77.08	81.68	77.51	67.50	72.16	78.24	62.92	69.74
1/2	78.8	82.08	80.41	70.13	67.51	68.79	75.13	60.42	66.97
1/4	93.64	76.67	80.00	65.88	70.00	67.81	77.14	45.00	56.84
1/8	73.73	78.33	75.96	65.84	66.67	66.25	71.03	42.92	53.51
1/16	76.19	66.67	71.11	58.33	64.17	61.14	57.07	48.75	52.58
1/32	75.54	57.92	65.56	60.27	56.25	58.23	72.72	23.33	35.33
DU	64.16	65.25	64.71	62.96	59.32	61.08			

We use gradually-reduced target-library training data (1 for all data, 1/2, 1/4, ..., 1/32) to fine-tune the source-library-trained model. We also train a target-library model from scratch (i.e., with randomly initialized model parameters) using the same proportion of the target-library training data for comparison. We also use the source-library-trained

TABLE 8: Matplotlib (MPL) or NumPy (NP) → Pandas (PD)

	MPL→PD			NP→PD			PD		
	Prec	Recall	F1	Prec	Recall	F1	Prec	Recall	F1
1/1	84.97	89.62	87.23	86.86	87.61	87.23	80.43	85.30	82.80
1/2	86.18	84.43	85.30	83.97	83.00	83.48	88.01	74.06	80.43
1/4	81.07	87.61	84.21	81.07	82.70	81.88	76.50	76.95	76.72
1/8	87.54	80.98	84.13	85.76	76.37	80.79	69.30	83.29	75.65
1/16	82.04	78.96	80.47	82.45	75.79	78.98	84.21	50.72	53.31
1/32	81.31	75.22	78.14	81.43	72.05	76.45	84.25	30.84	45.14
DU	71.65	40.06	51.39	75.00	35.45	48.14			

TABLE 9: Averaged Matplotlib (MPL) or NumPy (NP)→Pandas (PD)

	MPL→PD			NP→PD		
	Prec	Recall	F1	Prec	Recall	F1
1/16	83.33	77.81	80.43	83.39	75.22	79.09
1/32	79.50	73.78	76.53	86.15	71.15	78.30

model directly without any fine-tuning (i.e. 0/1 target-library data) as a baseline. We also randomly select 1/16 and 1/32 training data for *NumPy* → *Pandas* and *Matplotlib* → *Pandas* for 10 times, and train the model for each time. Then we calculate the averaged precision and recall values. Based on the averaged precision and recall values, we calculate the averaged F1-score.

Results: Table 6, Table 7 and Table 8 show the experiment results of the six pairs of transfer learning. Table 9 shows the averaged experiment results of *NumPy* → *Pandas* and *Matplotlib* → *Pandas* with 1/16 and 1/32 training data. Acronyms stand for: MPL (Matplotlib), NP (NumPy), PD (Pandas), DU (Direct Use). The last column is the results of training the target-library model from scratch. We can see that:

- *Transfer learning can produce a better-performance target-library model than training the target-library model from scratch with the same proportion of training data*. This is evident as the F1-score of the target-library model obtained by fine-tuning the source-library-trained model is higher than the F1-score of the target-library model trained from scratch in all the transfer settings but PD → MPL at 1/1. Especially for *NumPy*, the *NumPy* model trained from scratch with all *Numpy* training data has F1-score 69.74. However, the *NumPy* model transferred from the *Matplotlib* model has F1-score 81.68 (17.1% improvement). For the *Matplotlib* → *Nump* transfer, even with only 1/16 *Numpy* training data for fine-tuning, the F1-score (71.11) of the transferred *NumPy* model is still higher than the F1-score of the *NumPy* model trained from scratch with all *Numpy* training data. This suggests that transfer learning may boost the model performance even for the difficult dataset. Such performance boost by transfer learning has also been observed in many studies [14], [44], [45]. The reason is that a target-library model can "reuse" much knowledge in the source-library model, rather than having to learning completely new knowledge from scratch.
- *Transfer learning can reduce the amount of target-library training data required to obtain a high-quality model*. For four out of six transfer-learning pairs (NP → MPL, MPL → NP, NP → PD, MPL → PD), the reduction ranges from 50% (1/2) to 87.5% (7/8) while the resulting target-library model still has better F1-score than the target-library model trained from scratch using all target-library training data. If we allow a small degradation of F1-score (e.g., 3% of the F1-sore for the target-library model trained from scratch using all target-library training data), the reduction can go up to 93.8% (15/16). For example, for *Matplotlib* → $Pandas$, using 1/16 *Pandas* training data (i.e., only about 20 posts) for fine-tuning, the obtained *Pandas* model still has F1-score 80.47.
- *Transfer learning is very effective in few-shot training settings*. Few-shot training refers to training a model with only a very small amount of data [46]. In our experiments, using 1/32 training data, i.e., about 10 posts for transfer learning, the F1-score of the obtained target-library model is still improved a few points for NP → MPL, PD → MPL and MPL → NP, and is significantly improved for MPL → PD (52.9%) and NP → PD (58.3%), compared with directly reusing source-library-trained model without fine-tuning (DU row). Furthermore, the averaged results of NP → PD and MPL → PD for 10 randomly selected 1/16 and 1/32 training data are similar to the results in Table 8, and the variances are all smaller than 0.3, which shows our training data is unbiased. Although the target-library model trained from scratch still has reasonable F1-score with 1/2 to 1/8 training data, they become completely useless with 1/16 or 1/32 training data. In contrast, the target-library models obtained through transfer learning have significant better F1-score in such few shot settings. Furthermore, training a model from scratch using few-shot data may result in abnormal increase in precision (e.g., MPL at 1/16 and 1/32, NP at 1/32, PD at 1/32) or in recall (e.g., NP at 1/16), compared with training the model from scratch using more data. This abnormal increase in precision (or recall) comes with a sharp decrease in recall (or precision). Our analysis shows that this phenomenon is caused by the biased training data in few-shot settings. In contrast, the target-library models obtained through transfer learning can reuse knowledge in the source model and produce a much more balanced precision and recall (thus much better F1-score) even in the face of biased few-shot training data.
- *The effectiveness of transfer learning does not correlate with the quality of source-library-trained model*. Based on a not-so-high-quality source model (e.g., *NumPy*), we can still obtain a high-quality target model through transfer learning. For example, NP → PD results in a *Pandas* model with F1-score > 80 using only 1/8 of *Pandas* training data. On the other hand, a high-quality source model (e.g., *Pandas*) may not boost the performance of the target model. Among all 36 transfer settings, we have one such case, i.e., PD → MPL at 1/1. The *Matplotlib* model transferred from the *Pandas* model using all *Matplotlib* training data is slightly worse than the *Matplotlib* model trained from scratch using all training data. This can be attributed to the differences of API naming characteristics between the two libraries (see the last bullet).
- *The more similar the functionalities and the API-naming/mention characteristics between the two libraries are, the more effectiveness the transfer learning can be*. *Pandas* and *NumPy* have similar functionalities and API-naming/mention characteristics. For PD → NP and NP → PD, the target-library model is less than 5% worse in F1-score with 1/8 target-library training data, compared with the target-library model trained from scratch using

all training data. In contrast, PD → MPL has 6.9% drop in F1-score with 1/2 training data, and NP → MPL has 10.7% drop in F1-score with 1/4/ training data, compared with the *Matplotlib* model trained from scratch using all training data.

- *Knowledge about non-polysemous APIs seems easy to expand, while knowledge about polysemous APIs seems difficult to adapt*. *Matplotlib* has the least polysemous APIs (16%), and *Pandas* and *NumPy* has much more polysemous APIs. Both MPL → PD and MPL → NP have high-quality target models even with 1/8 target-library training data. The transfer learning seems to be able to add the new knowledge "common words can be APIs" to the target-library model. In contrast, NP → MPL requires at 1/2 training data to have a high-quality target model (F1-score 83.16), and for PD → MPL, the *Matplotlib* model (F1-score 80.58) by transfer learning with all training data is even slightly worse than the model trained from scratch (F1-score 82.7). These results suggest that it can be difficult to adapt the knowledge "common words can be APIs" learned from *NumPy* and *Pandas* to *Matplotlib* which does not have many common-word APIs.

Transfer learning can effectively boost the performance of target-library model with less demand on target-library training data. Its effectiveness correlates more with the similarity of library functionalities and and API-naming/mention characteristics than the initial quality of source-library model.

6.5 Effectiveness of Across-Language Transfer Learning (RQ4)

Motivation: In the RQ3, we investigate the transferability of API extraction model across three Python libraries. In the RQ4, we want to further investigate the transferability of API extraction model in a more challenging setting, i.e., across different programming languages and across libraries with very different functionalities.

Approach: For the source language, we choose *Python*, one of the most popular programming languages. We randomly select 200 posts in each dataset of the three Python libraries, and combine these 600 posts as the training data for Python API extraction model. For the target languages, we choose three other popular programming languages: *Java*, *JavaScript* and *C*. That is, we have three transfer-learning pairs in this RQ: *Python* → *Java*, *Python* → *JavaScript*, and *Python* → *C*. For the three target languages, we intentionally choose the libraries that support very different functionalities from the three Python libraries. For Java, we choose *JDBC* (an API for accessing relational database). For JavaScript, we choose *React* (a library for web graphical user interface). For C, we choose *OpenGL* (a library for 3D graphics). As described in Section 6.1.2, we label 600 posts for each target-language library for the experiments. As in the RQ3, we use gradually-reduced target-language training data to fine-tune the source-language-trained model.

Results: Table 10, Table 11 and Table 12 show the experiment results for the three across-language transfer-learning pairs. These three tables use the same notation as Table 6, Table 7 and Table 8. We can see that:

- *Directly reusing the source-language-trained model on the target-language text produces unacceptable performance*. For the three Python libraries, directly reusing a source-library-trained model on the text of a target-library may still have reasonable performance, for example, NP → MPL (F1-score 69.16), PD → MPL (64.0), MPL → NP (64.71). However, for the three across-language transfer-learning pairs, the F1-score of direct model reuse is very low: *Python* → *Java* (34.91), *Python* → *JavaScript* (33.69), *Python* → *C* (35.95). These results suggest that there are still certain level of commonalities between the libraries with similar functionalities and of the same programming language, so that the knowledge learned in the source-library model may be largely reused in the target-library model. In contrast, the libraries of different functionalities and programming languages have much fewer commonalities, which makes it infeasible to directly deploy a source-language-trained model to the text of another language. In such cases, we have to either train the model for each language or library from scratch, or we may exploit transfer learning to adapt a source-language-trained model to the target-language text.
- *Across-language transfer learning holds the same performance characteristics as within-language transfer learning, but it demands more target-language training data to obtain a high-quality model than within-language transfer learning*. For *Python* → *JavaScript* and *Python* → *C*, with as minimum as 1/16 target-language training data, we can obtain a fine-tuned target-language model whose F1-score can double that of directly reusing the Python-trained model to JavaScript or C text. For *Python* → *Java*, fine-tuning the Python-trained model with 1/16 Java training data boosts the F1-score by 50%, compared with directly applying the Python-trained model to Java text. For all the transfer-learning settings in Table 10, Table 11 and Table 12, the fine-tuned target-language model always outperforms the corresponding target-language model trained from scratch with the same amount of target-language training data. However, it requires at least 1/2 target-language training data to fine-tune a target-language model to achieve the same level of performance as the target-language model trained from scratch with all target-

TABLE 10: Python → Java

	Python→Java			Java		
	Prec	Recall	F1	Prec	Recall	F1
1/1	77.45	66.56	71.60	84.69	55.31	66.92
1/2	72.20	62.50	67.06	77.38	53.44	63.22
1/4	69.26	50.00	58.08	71.22	47.19	56.77
1/8	50.00	64.06	56.16	75.15	39.69	51.94
1/16	55.71	48.25	52.00	75.69	34.06	46.98
1/32	56.99	40.00	44.83	77.89	23.12	35.66
DU	44.44	28.75	34.91			

TABLE 11: Python → JavaScript

	Python→JavaScript			JavaScript		
	Prec	Recall	F1	Prec	Recall	F1
1/1	77.45	81.75	86.19	87.95	78.17	82.77
1/2	86.93	76.59	81.43	87.56	59.84	77.70
1/4	86.84	68.65	74.24	83.08	66.26	73.72
1/8	81.48	61.11	69.84	85.98	55.95	67.88
1/16	71.11	63.49	68.08	87.38	38.48	53.44
1/32	66.67	52.38	58.67	65.21	35.71	46.15
DU	51.63	25.00	33.69			

TABLE 12: Python → C

	Python→C			C		
	Prec	Recall	F1	Prec	Recall	F1
1/1	89.87	85.09	87.47	85.83	83.74	84.77
1/2	85.35	83.73	84.54	86.28	76.69	81.21
1/4	83.83	75.88	79.65	82.19	71.27	76.34
1/8	74.32	73.71	74.01	78.68	68.02	72.97
1/16	75.81	69.45	72.60	88.52	57.10	69.42
1/32	69.62	65.85	67.79	87.89	45.25	59.75
DU	56.49	27.91	35.95			

language training data. For the within-language transfer learning, achieving this result may require as minimum as 1/8 target-library training data.

Software text of different programming languages and of libraries with very different functionalities increases the difficulty of transfer learning, but transfer learning can still effectively boost the performance of the target-language model with 1/2 less target-language training data, compared with training the target-language model from scratch with all training data.

6.6 Threats to Validity

The major threat to internal validity is the API labeling errors in the dataset. In order to decrease errors, the first two authors first independently label the same data and then resolve the disagreements by discussion. Furthermore, in our experiments, we have to examine many API extraction results. We only spot a few instances of overlooked or erroneously-labeled API mentions. Therefore, the quality of our dataset is trustworthy.

The major threat to external validity is the generalization of our results and findings. Although we invest significant time and effort to prepare datasets, conduct experiments and analyze results, our experiments involve only six libraries of four programming languages. The performance of our neural architecture, and especially the findings on transfer learning, could be different with other programming languages and libraries. Furthermore, the software text in all the experiments comes from a single data source, i.e., Stack Overflow. Software text from other data sources may exhibit different API-mention and discussion-context characteristics, which may affect the model performance. In the future, we will reduce this threat by applying our approach to more languages/libraries and informal software text from other data sources (e.g., developer emails).

7 Related Work

APIs are the core resources for software development and API related knowledge is widely present in software texts, such as API reference documentation, Q&A discussions, bug reports. Researchers have proposed many API extraction methods to support various software engineering tasks, especially document traceability recovery [42], [47], [48], [49], [50]. For example, RecoDoc [3] extracts Java APIs from several resources and then recover traceability across different sources. Subramanian et al. [4] use code context information to filter candidate APIs in a knowledge base for an API mention in a partial code fragment. Christoph and Robillard [51] extracts sentences about API usage from Stack Overflow and use them to augment API reference documentation. These works focus mainly on the API linking task, i.e., linking API mentions in text or code to some API entities in a knowledge base. In terms of extracting API mentions in software text, they rely on rule-based methods, for example regular expressions of distinct orthographic features of APIs, such as camelcase, special characters (e.g., . or ()), and API annotations.

Several studies, such as Bacchelli et al [42], [43] and Ye et al. [2], show that regular expressions of distinct orthographic features are not reliable for API extraction tasks in informal texts, such as emails, Stack Overflow posts. Both variations of sentence formats and the wide presence of mentions of polysemous API simple names pose a great challenge for API extraction in informal texts. However, this challenge is generally avoided by considering only API mentions with distinct orthographic features in existing works [3], [51], [52].

Island parsing provides a more robust solution for extracting API mentions from texts. Using an island grammar, we can separate the textual content into constructs of interest (island) and the remainder (water) [53]. For example, Bacchelli et al. [54] uses island parsing to extract code fragments from natural language text. Rigby and Robillard [52] also use island parser to identify code-like elements that can potentially be APIs. However, these island parsers cannot effectively deal with mentions of API simple names that are not suffixed by (), such as *apply* and *series* in Fig. 1, which are very common writing forms of API methods in Stack Overflow discussions [2].

Recently, Ye et al. [10] proposes a CRF based approach for extracting methods of software entities, such as programming languages, libraries, computing concepts, and APIs from informal software texts. They report that extract API mentions is much more challenging than extracting other types of software entities. A follow-up work by Ye et al. [2] proposes to use a combination of orthographic, word-clusters and API gazetteer features to enhance the CRF's performance on API extraction tasks. Although these proposed features are effective, they require much manual effort to develop and tune. Furthermore, enough training data has to be prepared and manually labeled for applying their approach to the software text of each library to be processed. These overheads pose a practical limitation to deploying their approach to a larger number of libraries.

Our neural architecture is inspired by recent advances of neural network techniques for NLP tasks. For example, both RNNs and CNNs have been used to embed character-level features for question answering [55], [56], machine translation [57], text classification [58], and part-of-speech tagging [59]. Some researchers also use word embeddings and LSTMs for NER [35], [60], [61]. To the best of our knowledge, our neural architecture is the first machine learning based API extraction method that combines these proposals and customize them based on the characteristics of software texts and API names. Furthermore, the design of our neural architecture also takes into account the deployment overhead of the API methods for multiple programming languages and libraries, which has never been explored for API extraction tasks.

8 CONCLUSION

This paper presents a novel multi-layer neural architecture that can effectively learn important character-, word- and sentence-level features from informal software texts for extracting API mentions in text. The learned features has superior performance than human-defined orthographic features for API extraction in informal software texts. This makes our neural architecture easy to deploy, because the only input it requires is the texts to be processed. In contrast, existing machine learning based API extraction methods have to use additional hand-crafted features such as word clusters or API gazetteers, in order to achieve the performance close to that of our neural architecture. Furthermore, as the features are automatically learned from the input texts, our neural architecture is easy to transfer and fine-tune across programming languages and libraries. We demonstrate its transferability across three Python libraries and across four programming languages. Our neural architecture, together with transfer learning, makes it easy to train and deploy a high-quality API extraction model for multiple programming languages and libraries, with much less overall effort required for preparing training data and effective features. In the future, we will further investigate the performance and the transferability of our neural architecture in many other programming languages and libraries, moving towards real-world deployment of machine learning based API extraction methods.

REFERENCES

[1] C. Parnin, C. Treude, L. Grammel, and M.-A. Storey, "Crowd documentation: Exploring the coverage and the dynamics of api discussions on stack overflow."

[2] D. Ye, Z. Xing, C. Y. Foo, J. Li, and N. Kapre, "Learning to extract api mentions from informal natural language discussions," in *Software Maintenance and Evolution (ICSME), 2016 IEEE International Conference on*. IEEE, 2016, pp. 389–399.

[3] B. Dagenais and M. P. Robillard, "Recovering traceability links between an api and its learning resources," in *Software Engineering (ICSE), 2012 34th International Conference on*. IEEE, 2012, pp. 47–57.

[4] S. Subramanian, L. Inozemtseva, and R. Holmes, "Live api documentation," in *Proceedings of the 36th International Conference on Software Engineering*. ACM, 2014, pp. 643–652.

[5] 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.

[6] X. Chen, C. Chen, D. Zhang, and Z. Xing, "Sethesaurus: Wordnet in software engineering," *IEEE Transactions on Software Engineering*, 2019.

[7] H. Li, S. Li, J. Sun, Z. Xing, X. Peng, M. Liu, and X. Zhao, "Improving api caveats accessibility by mining api caveats knowledge graph," in *Software Maintenance and Evolution (ICSME), 2018 IEEE International Conference on*. IEEE, 2018.

[8] Z. X. D. L. X. W. Qiao Huang, Xin Xia, "Api method recommendation without worrying about the task-api knowledge gap," in *Automated Software Engineering (ASE), 2018 33th IEEE/ACM International Conference on*. IEEE, 2018.

[9] B. Xu, Z. Xing, X. Xia, and D. Lo, "Answerbot: automated generation of answer summary to developersź technical questions," in *Proceedings of the 32nd IEEE/ACM International Conference on Automated Software Engineering*. IEEE Press, 2017, pp. 706–716.

[10] D. Ye, Z. Xing, C. Y. Foo, Z. Q. Ang, J. Li, and N. Kapre, "Software-specific named entity recognition in software engineering social content," in *2016 IEEE 23rd International Conference on Software Analysis, Evolution, and Reengineering (SANER)*, vol. 1. IEEE, 2016, pp. 90–101.

[11] E. F. Tjong Kim Sang and F. De Meulder, "Introduction to the conll-2003 shared task: Language-independent named entity recognition," in *Proceedings of the seventh conference on Natural language learning at HLT-NAACL 2003-Volume 4*. Association for Computational Linguistics, 2003, pp. 142–147.

[12] R. Caruana, "Learning many related tasks at the same time with backpropagation," in *Advances in neural information processing systems*, 1995, pp. 657–664.

[13] A. Arnold, R. Nallapati, and W. W. Cohen, "Exploiting feature hierarchy for transfer learning in named entity recognition," *Proceedings of ACL-08: HLT*, pp. 245–253, 2008.

[14] J. Yosinski, J. Clune, Y. Bengio, and H. Lipson, "How transferable are features in deep neural networks?" in *Advances in neural information processing systems*, 2014, pp. 3320–3328.

[15] Z. X. Deheng Ye, Lingfeng Bao and S.-W. Lin, "Apireal: An api recognition and linking approach for online developer forums," *Empirical Software Engineering 2018*, 2018.

[16] C. dos Santos and M. Gatti, "Deep convolutional neural networks for sentiment analysis of short texts," in *Proceedings of COLING 2014, the 25th International Conference on Computational Linguistics: Technical Papers*, 2014, pp. 69–78.

[17] C. D. Santos and B. Zadrozny, "Learning character-level representations for part-of-speech tagging," in *Proceedings of the 31st International Conference on Machine Learning (ICML-14)*, 2014, pp. 1818–1826.

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

[19] C. Chen, X. Chen, J. Sun, Z. Xing, and G. Li, "Data-driven proactive policy assurance of post quality in community q&a sites," *Proceedings of the ACM on human-computer interaction*, vol. 2, no. CSCW, p. 33, 2018.

[20] I. Bazzi, "Modelling out-of-vocabulary words for robust speech recognition," Ph.D. dissertation, Massachusetts Institute of Technology, 2002.

[21] 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.

[22] C. Chen and Z. Xing, "Similartech: automatically recommend analogical libraries across different programming languages," in *2016 31st IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 2016, pp. 834–839.

[23] C. Chen, Z. Xing, and Y. Liu, "Whats spains paris? mining analogical libraries from q&a discussions," *Empirical Software Engineering*, vol. 24, no. 3, pp. 1155–1194, 2019.

[24] Y. Huang, C. Chen, Z. Xing, T. Lin, and Y. Liu, "Tell them apart: distilling technology differences from crowd-scale comparison discussions." in *ASE*, 2018, pp. 214–224.

[25] O. Levy and Y. Goldberg, "Neural word embedding as implicit matrix factorization," in *Advances in neural information processing systems*, 2014, pp. 2177–2185.

[26] D. Tang, F. Wei, N. Yang, M. Zhou, T. Liu, and B. Qin, "Learning sentiment-specific word embedding for twitter sentiment classification," in *Proceedings of the 52nd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, vol. 1, 2014, pp. 1555–1565.

[27] O. Levy, Y. Goldberg, and I. Dagan, "Improving distributional similarity with lessons learned from word embeddings," *Transactions of the Association for Computational Linguistics*, vol. 3, pp. 211–225, 2015.

[28] J. Pennington, R. Socher, and C. Manning, "Glove: Global vectors for word representation," in *Proceedings of the 2014 conference on empirical methods in natural language processing (EMNLP)*, 2014, pp. 1532–1543.

[29] I. Sutskever, O. Vinyals, and Q. V. Le, "Sequence to sequence learning with neural networks," in *Advances in neural information processing systems*, 2014, pp. 3104–3112.

[30] 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, p. 32, 2017.

[31] 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, pp. 414–421.
[32] X. Wang, C. Chen, and Z. Xing, "Domain-specific machine translation with recurrent neural network for software localization," *Empirical Software Engineering*, pp. 1–32, 2019.
[33] C. Chen, Z. Xing, Y. Liu, and K. L. X. Ong, "Mining likely analogical apis across third-party libraries via large-scale unsupervised api semantics embedding," *IEEE Transactions on Software Engineering*, 2019.
[34] M. Schuster and K. K. Paliwal, "Bidirectional recurrent neural networks," *IEEE Transactions on Signal Processing*, vol. 45, no. 11, pp. 2673–2681, 1997.
[35] Z. Huang, W. Xu, and K. Yu, "Bidirectional lstm-crf models for sequence tagging," *arXiv preprint arXiv:1508.01991*, 2015.
[36] S. Hochreiter and J. Schmidhuber, "Long short-term memory," *Neural computation*, vol. 9, no. 8, pp. 1735–1780, 1997.
[37] H. Sak, A. Senior, and F. Beaufays, "Long short-term memory recurrent neural network architectures for large scale acoustic modeling," in *Fifteenth annual conference of the international speech communication association*, 2014.
[38] Y. Bengio, "Deep learning of representations for unsupervised and transfer learning," in *Proceedings of ICML Workshop on Unsupervised and Transfer Learning*, 2012, pp. 17–36.
[39] S. J. Pan, Q. Yang *et al.*, "A survey on transfer learning."
[40] N. Srivastava, G. Hinton, A. Krizhevsky, I. Sutskever, and R. Salakhutdinov, "Dropout: A simple way to prevent neural networks from overfitting," *The Journal of Machine Learning Research*, vol. 15, no. 1, pp. 1929–1958, 2014.
[41] D. P. Kingma and J. Ba, "Adam: A method for stochastic optimization," *arXiv preprint arXiv:1412.6980*, 2014.
[42] A. Bacchelli, M. Lanza, and R. Robbes, "Linking e-mails and source code artifacts," in *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering-Volume 1*. ACM, 2010, pp. 375–384.
[43] A. Bacchelli, M. D'Ambros, M. Lanza, and R. Robbes, "Benchmarking lightweight techniques to link e-mails and source code," in *Reverse Engineering, 2009. WCRE'09. 16th Working Conference on*. IEEE, 2009, pp. 205–214.
[44] S. J. Pan and Q. Yang, "A survey on transfer learning," *IEEE Transactions on knowledge and data engineering*, vol. 22, no. 10, pp. 1345–1359, 2010.
[45] M. Oquab, L. Bottou, I. Laptev, and J. Sivic, "Learning and transferring mid-level image representations using convolutional neural networks," in *Computer Vision and Pattern Recognition (CVPR), 2014 IEEE Conference on*. IEEE, 2014, pp. 1717–1724.
[46] S. Ravi and H. Larochelle, "Optimization as a model for few-shot learning," 2016.
[47] A. Marcus, J. Maletic *et al.*, "Recovering documentation-to-source-code traceability links using latent semantic indexing," in *Software Engineering, 2003. Proceedings. 25th International Conference on*. IEEE, 2003, pp. 125–135.
[48] H.-Y. Jiang, T. N. Nguyen, X. Chen, H. Jaygarl, and C. K. Chang, "Incremental latent semantic indexing for automatic traceability link evolution management," in *Proceedings of the 2008 23rd IEEE/ACM International Conference on Automated Software Engineering*. IEEE Computer Society, 2008, pp. 59–68.
[49] W. Zheng, Q. Zhang, and M. Lyu, "Cross-library api recommendation using web search engines," in *Proceedings of the 19th ACM SIGSOFT symposium and the 13th European conference on Foundations of software engineering*. ACM, 2011, pp. 480–483.
[50] Q. Gao, H. Zhang, J. Wang, Y. Xiong, L. Zhang, and H. Mei, "Fixing recurring crash bugs via analyzing q&a sites (t)," in *Automated Software Engineering (ASE), 2015 30th IEEE/ACM International Conference on*. IEEE, 2015, pp. 307–318.
[51] C. Treude and M. P. Robillard, "Augmenting api documentation with insights from stack overflow," in *Software Engineering (ICSE), 2016 IEEE/ACM 38th International Conference on*. IEEE, 2016, pp. 392–403.
[52] P. C. Rigby and M. P. Robillard, "Discovering essential code elements in informal documentation," in *Proceedings of the 2013 International Conference on Software Engineering*. IEEE Press, 2013, pp. 832–841.
[53] L. Moonen, "Generating robust parsers using island grammars," in *Reverse Engineering, 2001. Proceedings. Eighth Working Conference on*. IEEE, 2001, pp. 13–22.
[54] A. Bacchelli, A. Cleve, M. Lanza, and A. Mocci, "Extracting structured data from natural language documents with island parsing," in *Automated Software Engineering (ASE), 2011 26th IEEE/ACM International Conference on*. IEEE, 2011, pp. 476–479.
[55] Y. Kim, Y. Jernite, D. Sontag, and A. M. Rush, "Character-aware neural language models." in *AAAI*, 2016, pp. 2741–2749.
[56] D. Lukovnikov, A. Fischer, J. Lehmann, and S. Auer, "Neural network-based question answering over knowledge graphs on word and character level," in *Proceedings of the 26th international conference on World Wide Web*. International World Wide Web Conferences Steering Committee, 2017, pp. 1211–1220.
[57] W. Ling, I. Trancoso, C. Dyer, and A. W. Black, "Character-based neural machine translation," *arXiv preprint arXiv:1511.04586*, 2015.
[58] X. Zhang, J. Zhao, and Y. LeCun, "Character-level convolutional networks for text classification," in *Advances in neural information processing systems*, 2015, pp. 649–657.
[59] C. D. Santos and B. Zadrozny, "Learning character-level representations for part-of-speech tagging," in *Proceedings of the 31st International Conference on Machine Learning (ICML-14)*, 2014, pp. 1818–1826.
[60] J. P. Chiu and E. Nichols, "Named entity recognition with bidirectional lstm-cnns," *arXiv preprint arXiv:1511.08308*, 2015.
[61] G. Lample, M. Ballesteros, S. Subramanian, K. Kawakami, and C. Dyer, "Neural architectures for named entity recognition," *arXiv preprint arXiv:1603.01360*, 2016.

Suyu Ma is a research assistant in the Faculty of Information Technology, at Monash University. He has research interest in the areas of software engineering, Deep learning and Human-computer Interaction. He is currently focusing on improving the usability and accessibility of mobile application. He received the B.S., and M.S., degrees from Beijing Technology and Business University and the Australian National University in 2016 and 2018, respectively. And he will be a PhD student at Monash University in 2020, under the supervision of Chunyang Chen.

Zhenchang Xing is a Senior Lecturer in the Research School of Computer Science, Australian National University. Previously, he was an Assistant Professor in the School of Computer Science and Engineering, Nanyang Technological University, Singapore, from 2012-2016. Before joining NTU, Dr. Xing was a Lee Kuan Yew Research Fellow in the School of Computing, National University of Singapore from 2009-2012. Dr. Xings current research area is in the interdisplinary areas of software engineering, human-computer interaction and applied data analytics. Dr. Xing has over 100 publications in peer-refereed journals and conference proceedings, and have received several distinguished paper awards from top software engineering conferences. Dr. Xing regularly serves on the organization and program committees of the top software engineering conferences, and he will be the program committee co-chair for ICSME2020.

Chunyang Chen is a lecturer (Assistant Professor) in Faculty of Information Technology, Monash University, Australia. His research focuses on software engineering, deep learning and human-computer interaction. He has published over 16 papers in referred journals or conferences, including Empirical Software Engineering, ICSE, ASE, CSCW, ICSME, SANER. He is a member of IEEE and ACM. He received ACM SIGSOFT distinguished paper award in ASE 2018, best paper award in SANER 2016, and best tool demo in ASE 2016. https://chunyang-chen.github.io/

Cheng Chen received his Bachelor degree in Software Engineering from Northwest University (China), and received Master degree of Computing (major in Artificial Intelligence) from the Australian National University. He did some internship projects about Natural Language Processing at CSIRO from 2016 to 2017. He worked as a research assistant supervised by Dr. Zhenchang Xing at ANU in 2018. Cheng currently works in PricewaterhouseCoopers (PwC) Firm as a senior algorithm engineer of Natural Language Processing and Data Analyzing. Cheng is interested in Named Enti ty Extraction, Relation Extraction, Text summarization and parallel computing. He is working on knowledge engineering and transaction for NLP tasks.

Lizhen Qu is a research fellow in the Dialogue Research lab, in the Faculty of Information Technology, at Monash University. He has extensive research experience in the areas of natural language processing, multimodal learning, deep learning, and Cybersecurity. He is currently focusing on information extraction, semantic parsing, and multimodal dialogue systems. Prior to joining Monash University, he worked as a research scientist at Data61/CSIRO, where he led and participated in several research and industrial projects, including Deep Learning for Text and Deep Learning for Cyber.

Guoqiang Li is now an associate professor in school of software, Shanghai Jiao Tong University, and a guest associate professor in Kyushu University. He received the B.S., M.S., and Ph.D. degrees from Taiyuan University of Technology, Shanghai Jiao Tong University, and Japan Advanced Institute of Science and Technology in 2001, 2005, and 2008, respectively. He worked as a postdoctoral research fellow in the graduate school of information science, Nagoya University, Japan, during 2008-2009, as an assistant professor in the school of software, Shanghai Jiao Tong University, during 2009-2013, and as an academic visitor in the department of computer science, University of Oxford during 2015-2016. His research interests include formal verification, programming language theory, and data analytics and intelligence. He published more than 40 researches papers in the international journals and mainstream conferences, including TDSC, SPE, TECS, IJFCS, SCIS, CSCW, ICSE, FORMATS, ATVA, etc.