ARCLIN: Automated API Mention Resolution for Unformatted Texts

Yintong Huo
The Chinese University of Hong Kong
Hong Kong, China
ythuo@cse.cuhk.edu.hk

Yuxin Su*
School of Software Engineering
Sun Yat-sen University
Zhuhai, China
suyx35@mail.sysu.edu.cn

Hongming Zhang
The Hong Kong University of Science and Technology
Hong Kong, China
hzhangal@cse.ust.hk

Michael R. Lyu
The Chinese University of Hong Kong
Hong Kong, China
lyu@cse.cuhk.edu.hk

ABSTRACT

Online technical forums (e.g., StackOverflow) are popular platforms for developers to discuss technical problems such as how to use a specific Application Programming Interface (API), how to solve the programming tasks, or how to fix bugs in their code. These discussions can often provide auxiliary knowledge of how to use the software that is not covered by the official documents. The automatic extraction of such knowledge may support a set of downstream tasks like API searching or indexing. However, unlike official documentation written by experts, discussions in open forums are made by regular developers who write in short and informal texts, including spelling errors or abbreviations. There are three major challenges for the accurate APIs recognition and linking mentioned APIs from unstructured natural language documents to an entry in the API repository: (1) distinguishing API mentions from common words; (2) identifying API mentions without a fully qualified name; and (3) disambiguating API mentions with similar method names but in a different library. In this paper, to tackle these challenges, we propose an ARCLIN tool, which can effectively distinguish and link APIs without using human annotations. Specifically, we first design an API recognizer to automatically extract API mentions from natural language sentences by a Conditional Random Field (CRF) on the top of a Bi-directional Long Short-Term Memory (Bi-LSTM) module, then we apply a context-aware scoring mechanism to compute the mention-entry similarity for each entry in an API repository. Compared to previous approaches with heuristic rules, our proposed tool without manual inspection outperforms by 8% in a high-quality dataset Py-mention, which contains 558 mentions and 2,830 sentences from five popular Python libraries. To our best knowledge, ARCLIN is the first approach to achieve full automation of API mention resolution from unformatted text without manually collected labels.

KEYWORDS

API, API disambiguation, text mining

ACM Reference Format:
Yintong Huo, Yuxin Su, Hongming Zhang, and Michael R. Lyu. 2022. ARCLIN: Automated API Mention Resolution for Unformatted Texts. In *44th International Conference on Software Engineering (ICSE '22), May 21–29, 2022, Pittsburgh, PA, USA.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3510003.3510158

1 INTRODUCTION

Application Programming Interface (API) is an essential component for programming. Developers use APIs to interact with a programming language or a software library. However, as a library contains thousands of APIs (e.g., PyTorch v1.8 has over 2,400 APIs) and there are hundreds of popular libraries in a language, it is impossible for developers to be familiar with all APIs. Therefore, developers are used to discussing programming-related questions in the online technical forum when they face troubles in programming tasks. One of the most popular forums, StackOverflow, contains over 20 million questions and 14 million users[1]. It motivates researchers to explore how to identify knowledge in open forums to assist developers in many aspects, such as API recommendation [37], API misuse detection [29, 30], and document augmentation [35].

The foundation of the above tasks is recognizing and identifying API mentions from an unstructured natural language. Conventionally, researchers tried to use rule-based methods to solve the task. For example, Bacchelli et al. [2], Treude and Robillard [35] identified API elements in texts by a set of regular expressions. Huang et al. [18] chose a hyperlink in each StackOverflow post and used regular expressions to detect API entities. They also analyzed whether the text in HTML <code> tag can match the API names in the API repositories. Li et al. [21] detected APIs by checking whether the token of a sentence can match or partially match the name of an API by conducting minor modifications. Ren et al. [30] kept API mentions only in HTML <code> elements.

However, these rule-based methods do not consider the short and informal nature of forum discussions, falling short in mining APIs

*Corresponding author (suyx35@mail.sysu.edu.cn).

ICSE '22, May 21–29, 2022, Pittsburgh, PA, USA

ACM ISBN 978-1-4503-9221-1/22/05…$15.00
https://doi.org/10.1145/3510003.3510158

[1]The data dump is retrieved in September 1st, 2021.

in certain scenario. Typically, a forum may contain a large number of unprofessional developers with different technical backgrounds, who share the knowledge and information in their own writing styles. As a result, the API mentions could be in different formats. For example, previous study [33] concluded from StackOverflow posts that 47% of the API elements are not included with the HTML <code> tag. Such inconsistency causes different kinds of ambiguity when we recognize and identify APIs. In this paper, we categorize these ambiguities into the following three types.

The first one is *common-word ambiguity*, referring to the ambiguity between common words and API mentions [40]. Traditionally, API name is composed of punctuations, brackets, and upper case letters; however, sometimes developers only write the API's method name in their answers, causing the difficulty of distinguishing it from common words. The first group in the Table 1 illustrates examples of this problem. Even if two sentences are all mentioning the word *view*, the first sentence use *view* to refer to the API *torch.view()* whereas the second one use *view* as a common verb. Regular expressions fail in discriminating such API mentions with common words. Previous work [40] revealed that 35.1% of the token *apply* in StackOverflow posts tagged with Pandas actually referred to an API mention.

The second one is *morphological ambiguity*, which is because developers rarely write down the full API name that can be perfectly matched with an API name in the library. Research on StackOverflow [5] concludes that morphological mentions, which include abbreviations, synonyms, and misspellings, are quite often in informal discussions. Four examples in the second group of Table 1 demonstrate the morphological variations. In the first three sentences, the API *numpy.reshape()* was mentioned by replacing *numpy* with its abbreviation *np*, omitting library name, and using the customized variable name, respectively. The fourth sentence talks about the *torch.nn.Conv2d* and *torch.nn.Conv3d* APIs, but includes neither the library/module/class name nor the correct case (i.e., use *conv2d* instead of *Conv2d*).

The third type is *reference ambiguity*, which happens if the API lists contain various third-party libraries. The third group in Table 1 provides two instances of this problem. Even if both PyTorch library and Tensorflow library contain the API method *flatten()*, we could characterize what the mentions refer to based on their sentence contexts (i.e., the first sentence mentions *"keras"* module whereas the second one mentions *"PyTorch"*). It is often the case that developers do not explicitly point out the specific library in their mentions, but such information can be derived from other words in the context.

Due to the above ambiguities, traditional information retrieval techniques cannot be effectively employed. Dagenais and Robillard [7] applied a set of filtering heuristics to tackle the second challenge, but they failed in resolving common words ambiguity due to the shortcoming of regular expressions. The above challenges become more difficult if we apply the API mining task into unformatted sentences. Such free text does not contain any <code> tags, so detecting API in this scenario is even harder. However, it is a non-negligible problem, since, in other scenarios (e.g., emails), we cannot use HTML tags. To make our research applicable for a broader application, we focus on mining APIs from free text. Although the most recent work [40] claimed to distinguish API mentions from common words, they stored <code> tags and code snippets in <pre> <code> tags from StackOverflow posts, instead of mining from free texts. Thus, their approach cannot be extended to general scenarios.

In this paper, to overcome the aforementioned ambiguity challenges, we propose a new API mining approach named ARCLIN (**A**PI **R**ecognition and **C**ontextual **LIN**king), which recognizes and identifies API mentions from natural language descriptions to a set of APIs without any human-annotated labels or handcrafted rules. Our model is made up of an API recognizer that finds API mentions in free texts, and a contextual API linker that links API mentions to the correct API they refer to. Specifically, our API recognizer extensively deals with the first common word ambiguity by considering the context information in sentence-level around an API mention. For the words that are predicted to be an API mention, a library predictor inside the API linker predicts the related library to the sentence, restricting ARCLIN to link APIs in the predicted library, which resolves the reference ambiguity. The similarity function in the API linker compares API mention with every entry in the API repository, considering both spelling similarity and lexical similarity, so minor morphological changes will not affect the linking result. To the best of our knowledge, ARCLIN is the first approach that can automatically cope with these challenges above.

Considering the numerous number of APIs in the real world, it is impractical to ask annotators to label such a large scale of data. To avoid this labor-intensive process, we design ARCLIN to be free from any human annotation in the training process by exploiting natural labels in the training set. Unlike human-labeled data, the automated labels may contain errors, but our API linker in the next step provides a strict selection to address this problem. To evaluate the effectiveness of ARCLIN, we annotate a test set, which contains 2,948 sentences with 563 mentions under five popular third-party libraries. On average, ARCLIN achieves 78.26%, 73.53% and 75.82% in precision, recall and F1 score, respectively. The promising results indicate that, even though our approach does not need any human-annotated labels, it outperforms the current state-of-the-art baseline trained with labeled data.

To sum up, the main contributions of this paper are threefold:

- To our best knowledge, we are the first to design an unlabeled approach focusing on API recognizing and linking in unformatted text corpora.
- We build an API contextual linker, making the model automatically link API mentions to an API repository, taking the sentence context into account.
- The experiment results show ARCLIN can discover traceability links between APIs and the repository more accurately, comparing with state-of-the-art baseline models. The code and dataset are released[2].

2 PROBLEM STATEMENT

In this section, we first introduce the main concepts used in this paper in Section 2.1 and then provide a formal definition of the task in Section 2.2.

[2]Please find the resources in https://github.com/YintongHuo/ARCLIN.

Table 1: Three main challenges for API mining in unformatted texts, Blue words refers to API mentions and Red words refers to common words.

Question ID	Sentence	API
#66952125	So far I managed to use **view** once in my first very simple project...	torch.view()
#59905234	You can use .numpy() to **view** the internal data...	None
#42233297	Simply reshaping the by **np.reshape(data,(5000,3,32,32))** would not work.	numpy.reshape()
#41518351	The **.reshape()** method (of ndarray) returns the reshaped array.	numpy.reshape()
#47477945	I have tried **a.reshape(3,4)** in for the numpy array but nothing is producing what I want.	numpy.reshape()
#65103822	I would like to understand the difference between **conv2d** and **conv3d** in PyTorch.	torch.nn.Conv2d, torch.nn.Conv3d
#47532162	I want to use the keras layer **Flatten()** or Reshape((-1,)) at the end of my model...	tensorflow.keras.layers.Flatten()
#60115633	But in PyTorch, **flatten()** is an operation on the tensor.	torch.Tensor.flatten()

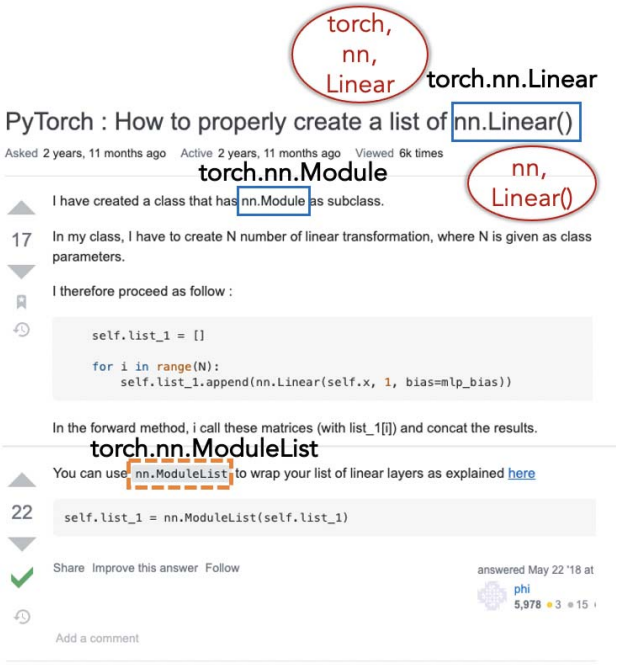

Figure 1: A screenshot of one StackOverflow post.

2.1 Terminology

API mining is the task of recognizing API mentions from free texts and linking the recognized API mentions to the corresponding API repositories. Figure 1 is a screenshot of a StackOverflow post[3], we use this screenshot to illustrate the concepts used in this paper. Here, an *API* could be a class name, a method name, or an attribute of a class. The term *free text* (also called *unformatted text*) refers to the text without any HTML tags (e.g., <code>). Orange dash box in Figure 1 shows an example of <code> usage. An *API mention* in texts is a token appearing in the free text that refers to a specific API in the repository. Blue box in the figure shows two API mentions (i.e., *nn.Linear()* and *nn.Module*)[4]. An *API repository* is a collection of all entire qualified API names. *Entire qualified name* is the exact API's name shown in its official website (e.g., *torch.nn.Linear*, *torch.nn.Module*). Each API's name in this repository is called an *entry*. An entry is composed of sub-fields, splitted by ".", which are called entities, the entities of *nn.Linear()* and *torch.nn.Linear* are shown in nearby red circles.

[3]The entire page is in https://stackoverflow.com/questions/50463975/pytorch-how-to-properly-create-a-list-of-nn-linear.
[4]*nn.ModuleList* in Orange box is also an API mention after removing the <code> tag.

2.2 Task Description

Given a natural language sentence S in free text and an API repository $D = \{D_1, D_2, ..., D_n\}$, where D_i refers to an entry in the repository, the API mining task is to link API mentions to an entry in the API repository. The task involves two phases: (1) recognizing API mentions in the sentence; (2) linking API mentions to the corresponding entries in the API repository. In practice, we first tokenize S into a token list $[token_0, token_1, ..., token_n]$, then $\forall 0 \leq i \leq n$, we determine whether $token_i$ refers to the element $D_j \in D$.

3 APPROACH

In this section, we introduce our approach, including data preparation, an API recognizer, and a contextual API linker. Figure 2 shows the overall framework of ARCLIN. To begin with, sentences are fed into the API recognizer to uncover API mentions. Specifically, a context encoder is applied to acquire contextual embeddings of tokens by a bidirectional Long-Short Term Memory (LSTM) network, then these representations are decoded via a Conditional Random Field (CRF). The tokens which decoded as API mentions are sent to the API linker. Next, an API linker is designed for discovering the most possible matched entry in the repository. To do so, we first generate a series of candidates by heuristic rules, then a library predictor narrows down the candidates by specifying a library. After that, we use an integrated scoring function to rank <mention, entry> pairs. Finally, the candidate with the highest similarity above the threshold will be chosen as a link.

3.1 Data Preparation

3.1.1 Text Corpus. Given some libraries, we crawl all questions tagged with at least one of the given libraries from an online technical forum. Besides questions and answers, Zhang et al. [43] revealed that the majority of comments were also informative as they provided a supplementary view to the answer. Therefore, for each question-answering thread, we crawl the question, all answers, and their comments. We discard code snippets in <pre> <code> but

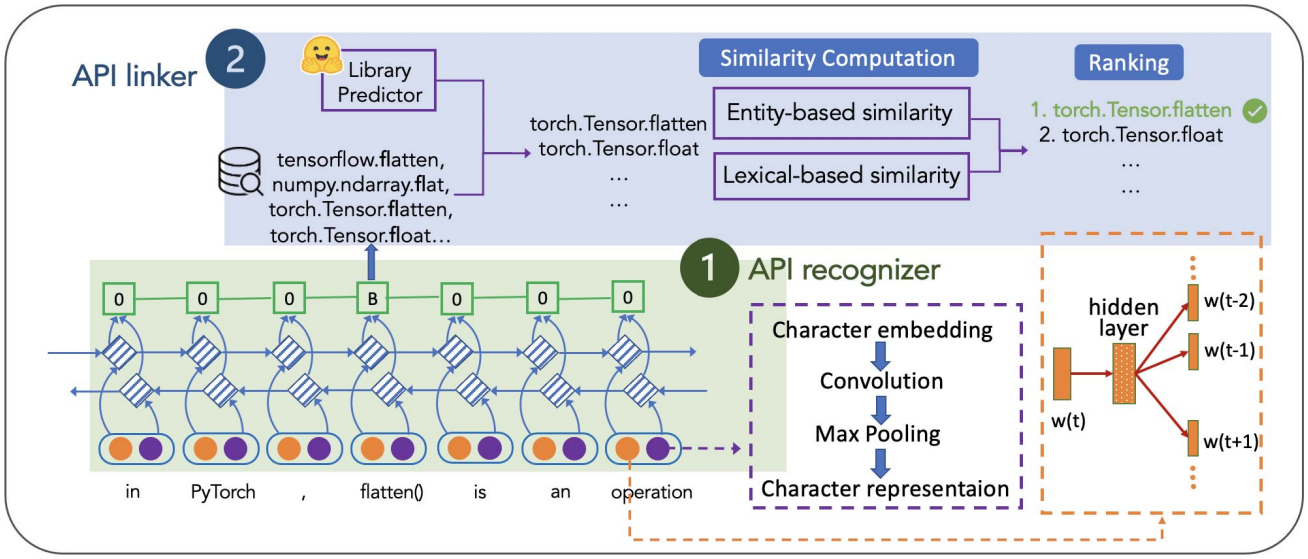

Figure 2: The framework of ARCLIN.

keep contents in <code> when it appears in a natural language sentence.

StackOverflow users highlight API mentions in a natural language sentence by <code> tags. However, Tabassum et al. [33] shows that 47% of the code mentions are not indicated with this tag. If we only rely on the tags to do the mention detection, we will miss a large number of mentions. Moreover, it is observed that contents in code tags can be noisy, many non-code elements (e.g., variables name, key points, or user name) are also highlighted by code tags [33]. To address this noisiness issue, as well as to generalize to other scenarios that cannot use <code> tags (e.g., emails), we remove all <code> tags in sentences and the markdown markers to make this task more similar to the real application. After collecting the data, we tokenize all sentences with the NLTK [3] sentence parser. As a result, we obtain a set of parsed sentences in free text.

3.1.2 Repository Construction. For each given library, we crawl all APIs with their entire qualified names from official documentations. For instance, the APIs in the PyTorch library include methods (e.g., *torch.Tensor.dim()*), functions (e.g., *torch.nn.functional.avg_pool1d()*), classes (e.g., *torch.nn.AdaptiveAvgPool1d*), and attributes such as *torch.backends.cudnn.enabled*. The API repository is made up of all crawled APIs' names.

3.1.3 Tokenizer. We adapt a software-specific tokenizer used in Ye et al. [41] and Ye et al. [42], which preserves the integrity of an API mention. Current popular tokenizers such as SpaCy [17], Stanford Parser [8], and NLTK all parse *numpy.shape()* into a token list of *["numpy.shape", "(", ")"]*, but the deployed software-specific tokenizer will treat *numpy.shape()* as a single token.

3.1.4 Inverse Document Frequency (IDF). IDF is a way to measure the importance of a word in a corpus. A word's IDF is disproportionate to the word's frequency. Given the assumption that if a word frequently occurs in a document, it may contain relatively less information, the formula for computing IDF for a word w is shown in Equation 1:

$$IDF(w) = log(\frac{\#Documents\_Number}{\#Document\_with\_w + 1}) \quad (1)$$

In this paper, we compute two types of IDF, IDF_{token} and IDF_{entity}. We use IDF_{token} to measure the token's importance in a corpus, where we regard each sentence as a document. For IDF_{entity}, we compute the entity's importance in the repository. We consider each entry is a document and its entities are words. For example, the document *numpy.reshape()* has two words: *"numpy"* and *"reshape()"*. Intuitively, since all Numpy APIs contain the entity *"numpy"*, its IDF value is relatively low.

3.2 Recognizer

The first step of mining APIs from the sentences is extracting API mentions without specifying which APIs they refer to. At this step, we propose an automatic API mention recognizer that prefers recall over precision to cover as many API mentions as possible. We first introduce an automatic approach to mine natural (but noisy) labels, then elaborate on architectures of the recognizer in the following subsections.

3.2.1 Automatic labeling. Traditional machine learning approaches label a large-scale training set to train classification models for the task. However, considering that there are enormous APIs even in one programming language, it is infeasible to obtain sufficient human-labeled data for all of them. There is a need to devise an algorithm that escapes from any human annotation.

Previous studies [27, 34] show that prior external knowledge (i.e., API repository) was critical for good performance in identifying named entity in a sentence. Motivated by this, we use the following criteria (i.e., domain knowledge) to automatically annotate potential API mentions:

- If a token is exactly the entire qualified name (i.e., same as an entry in API repository), we regard the token as an API mention.
- Inspired from that users usually use "*()*" at the end of a token to represent an API method name or function name, we treat the token as an API mention if the token contains "*()*".
- Users also use "." (e.g., *numpy.shape()*, or *x.shape()*) when they mention an API; thus, we consider the token is an API mention if it contains ".". To distinguish such mentions from emoticon or punctuation, we require the token to consist of more than three characters.

Moreover, to address the common-word polysemy problem introduced in Section 1, we employ a data augmentation technique for each sentence with at least one API mention being detected. Specifically, we randomly replace the originally detected API mention with a new one only containing the last part of the name (e.g., *x.view* will be replaced with *view*). This data augmentation process forces the recognizer to learn contextual information of an API mention.

The self-labeling process inevitably introduces some noisy labels. For instance, even if the token *"python2.7"* consists ".", it is not an API mention; Besides that, a missing space between two sentences (e.g., ... plot 500 ellipses on a single graph.If you do ...) will generate the wrong label for the token *"graph.If"*. However, the proposed contextual linker is able to mitigate these noisy labels. After automatic labeling, we feed the self-labeled data as well as the augmented ones into our context encoder.

3.2.2 Context Encoder. Context encoder is responsible for acquiring contextual word embeddings in a sentence. The long short-term memory (LSTM) network has shown promising results in sequential labeling tasks [32], due to its strong ability to capture long-distance context information. The memory unit in LSTM enables it to generate the representation based on both the short-distance and long-distance context. In this work, we design an LSTM network to achieve the goal. A bi-directional LSTM (Bi-LSTM)[16] is specifically used for preserving both past and future information within a sentence.

The architecture of the constructed encoder is shown in Figure 2, where two granular-level features are considered. By doing so, the Bi-LSTM encoder simultaneously grasps word-level semantics and character-level details. Firstly, word embedding techniques are used to extract word-level semantics. Word embedding represents words as distributed vectors in a low-dimensional space so that words with similar semantic or syntactic meaning tend to be close in their vector space. Assuming that words present in a similar context have similar meanings, the common approach Skip-gram (Word2Vec) [25] learns word embeddings by predicting surrounding words given the central word. Similar to previous research [15, 18, 38], we train domain-specific word embeddings by Skip-gram on a domain corpus.

Secondly, as previous study [22] has shown that character-level representation is crucial to extract morphological evidence, we use this feature to alleviate the second morphological problem mentioned in Section 1. Besides, since developers write API mentions with customized variable names under different scenarios, deploying character-level embedding allows us to cope with unseen words, named the out-of-vocabulary (OOV) problem. In particular, we elicit character-level features from the architecture shown in blue-dotted rectangles in Figure 2, which incorporates one max pooling layer after a Convolutional Neural Network (CNN) is applied.

3.2.3 Tag Decoder. Given contextual word representations in a sentence, the tag decoder is used to determine whether the word is an API mention or just a common word. Inspired by previous sequence labeling works [20] in the natural language processing domain, we adopt a Conditional Random Field (CRF) to conduct the tag decoder on top of the text encoder (i.e., Bi-LSTM layer). By accurately obtaining structural dependencies among adjacent words in a sentence, the CRF module jointly predicts the tag of each word sequentially instead of predicting tags independently. In order to balance between API mention coverage and precision in predictions, we select Top_P paths with the highest confidence score as the result of the CRF layer. If the token in $K(0 \leq K \leq P)$ paths is predicted as an API mention, we treat the token as an API mention and feed it to the contextual linker.

3.3 Contextual Linker

Once we obtain the API mentions in the text, ARCLIN links the correct API mentions to an entry in the repository. The core idea behind this linker is a series of disambiguation methods. Specifically, we firstly select entries as candidates in the repository, then rank the similarity score of every <mention,entry> pair with the help of the mention's context information. Although the predicted API mentions may contain errors, the wrong mention will be hard to find an entry with a high similarity score. From this aspect, the noise introduced by the last step will not affect the final results.

3.3.1 Candidate Selection. To reduce the time complexity of comparing all entries in the repository with the API mention, we narrow the scope by listing a set of candidates. Inspired by the fact that, even though humans can make errors in spelling words, such misspelling is hardly seen at the beginning of the word. So do the developers. Given a mention, we directly compare its last part (i.e., last entity) and the last part of the entries in the repository. If the first two characters of the last entities are case-insensitive matching, we add the entry to a candidate list.

3.3.2 Library Predictor. As the third challenge aforementioned, similar API entries in different libraries bring difficulties to disambiguate the mention. An intuitive way is to take sentence-level semantics into consideration. To capture rich contextual information from sentences, we first train the most popular language model BERT [9] with all training sentences for each library. Then, one fully connected layer followed by a soft-max output layer is fine-tuned to predict the library of input sentences based on the semantic embedding produced by BERT.

3.3.3 Similarity Computation. Given an API mention m and its candidates e, we calculate the similarity score between the API mention and each candidate. Finally, we rank all candidates based on their similarity and select the most relevant candidate above the threshold. Basically, we compute similarity based on bag similarity. Given two bags of entities, M, E_i being split by "." from the API

mention m and a candidate $e_i \in e$, respectively, we compute the similarity from two aspects, lexical-similarity and entity-similarity.

Lexical Similarity. This step is motivated by the fact that sometimes developers make spelling errors in sentences, especially when the mentioned API name is long. However, even if we make a typo in some words, its lexical meaning (i.e., word representation learned from corpus) will not change.

Inspired by Huang et al. [18], we use the Equation 2 to calculate lexical-based similarity between mention entities M and entities of one candidate E_i.

$$Sim_L(M \rightarrow E_i) = \frac{\sum_{w \in M} sim(w, E_i) * IDF_{token}(w)}{\sum_{w \in M} IDF_{token}(w)}, \quad (2)$$

where $IDF_{token}(w)$ represents the IDF value of token w in the training data. $sim(w, E_i)$ refers to the maximum lexical similarity score between the element $w \in M$ and elements in set E_i. We calculate lexical similarity for pairs of entities by another word embedding model FastText [4]. Unlike Word2Vec, FastText is the embedding model that incorporates n-gram features of a token, so it solves the OOV problem. Inversely, we also compute the similarity $Sim_L(E_i \rightarrow M)$ by exchanging M and E_i in Equation 3. In the end, the overall lexical similarity is formulated through an arithmetic mean operation:

$$Sim_L(M, E_i) = \frac{Sim_L(M \rightarrow E_i) + Sim_L(E_i \rightarrow M)}{2}. \quad (3)$$

Entity Similarity. Jaccard similarity coefficient [19] is widely used in gauging how similar the two sets are. Given two bags of entities M, E_i, we formulate our weighted Jaccard similarity as Equation 4:

$$Sim_J(M, E_i) = \frac{\sum_{w \in (M \cap E_i)} IDF_{entity}(w)}{\sum_{w \in E_i} IDF_{entity}(w)}, \quad (4)$$

where $IDF_{entity}(w)$ represents the IDF value of entity w in the API repository. IDF provides a standard to measure the salience of a token. a higher IDF value represents that it appears more frequently, carrying lower information entropy. For instance, in our repository, tokens such as *nn*, *torch*, *numpy* contain a low IDF value since it is almost present in every entry, but tokens such as *AdaptiveMaxPool1d* and *binary_cross_entropy* deserve more attention, thus a high IDF value. Intuitively, instead of class or module names, we always use method names to clarify the mentioned API, which contains a higher IDF value. Such discriminative tokens contribute significantly to this Sim_J function, while missing a match in *nn* just makes a minor effect on the entity similarity score.

Overall Similarity. To sum up, the scoring function for calculating similarity is composed of a lexical similarity function and an entity similarity function. Given an API mention m and an entry e_i, the overall similarity is calculated by Equation 5:

$$Sim(m, e_i) = Sim_L(m, E_i) + Sim_J(m, E_i), \quad (5)$$

where $Sim_L(m, E_i)$ and $Sim_J(m, E_i)$ are defined above. To exclude the API mentions that are wrong predictions introduced from the recognizer, and the API mentions that refer to an API out of our repository, we eliminate the candidates $e_i \in e$ with lower $Sim(m, e_i)$ value than the similarity threshold S. Finally, we rank all remaining candidates and choose the $e_j \in e$ with the highest $Sim(m, e_j)$ as output.

Table 2: Statistics of API repository and Py-mention set.

Library	Version	#API	#Mention	#Sentence
PyTorch [12]	1.8.0	2,472	133	562
Tensorflow [14]	2.4.1	10,361	87	532
Pandas [13]	1.2.4	2,174	117	573
Numpy [11]	1.20	1,913	116	580
Matplotlib [10]	3.4.1	6,937	105	583
Sum	-	23,857	558	2,830

Table 3: Statistics of training set.

Library	#Sentence	#Autolabel	#Augmentation
PyTorch	150,000	11,057	27,077
Tensorflow	150,000	8,925	21,899
Pandas	150,000	10,537	25450
Numpy	150,000	11,501	27,941
Matplotlib	150,000	9,769	23,564
Sum	750,000	51,789	125,931

4 EXPERIMENTAL SETUP

In this section, we introduce the experimental setup details, including data collection, implementation details, and evaluation metrics.

4.1 Data Collection

4.1.1 Text Preparation. In this paper, we focus on five widely-used third-party libraries in Python: *Pytorch*, *Pandas*, *Tensorflow*, *Numpy*, *Matplotlib*. We crawl all questions tagged with at least one of the above libraries using Scrapy in StackOverflow. For each question-answer thread, we collect questions, all answers and their comments. Details of the data preprocessing method are described in Section 3.1.1.

4.1.2 API Repository. We construct an API repository containing all API in five chosen third-party libraries with their entire qualified names. We use Scrapy to crawl all APIs from their official websites. Information such as the version of each library, the number of APIs in each library is listed in Table 2. Considering that parentheses *"()"* are not the sign to differ APIs from each other, we remove all *"()"* at the end of the API entire qualified names (e.g., store *numpy.einsum* instead of *numpy.einsum()*).

4.1.3 Dataset. Considering all texts crawled from text preparation are too large to cope with, we randomly sample 150,000 sentences for each of the libraries and treat them as unlabeled training data. After applying self-labeling and data augmentation, we obtain 125,931 sentences for training the recognizer. The distributions of training data, automatically API labels, and augmentation results are shown in Table 3.

For the testing data, we randomly select 600 sentences from each library (without overlapping with the training data) and ask experts to annotate them. To ensure annotation quality, two invited experts both have more than four years of experience in Python development and are all familiar with five libraries. Considering that a long sentence is more likely to contain API mentions, we select the testing data sentences longer than ten tokens. During

the annotation, given the whole API repository, experts are asked to annotate whether each token in a sentence is referring to an API in the repository or not. If yes, they need to write down the entire qualified name of an API mention. We also ask annotators to throw away the sentence if they are not confident at what it refers to. In this way, we collect 2,830 sentences with 558 API mentions from five libraries in total, where their distributions along with the repository's distribution are shown in Table 2. Typical examples are below:

- If you don't want to export, please uncomment *plt.show() [matplotlib.pyplot.show()]* and remove ...
- I've usually gotten good performance out of numpy's *einsum [numpy.einsum()]* function and I like ...
- Here is a way to do it using *stack [torch.stack()]* or *unbind [torch.unbind()]*.

Here, *black italic fonts* indicates API mentions and *blue italic fonts* in brackets are the linked APIs in the repository (with entire qualified names).

4.2 Implementation Details

In the data preprocessing period, we train a skip-gram Word2Vec model based on our corpus with gensim [28]. We also train a FastText word embedding model with gensim [28] for computing <mention, entry> pair lexical-based similarity. The embedding size for Word2Vec and FastText models are set to 300. Two models are trained for ten epochs[5]. IDF_{token} and IDF_{entity} are trained on the training data.

For the API recognition part, we use an open-source natural sequence labeling tool from [39] as the implementation and train the recognizer on the augmented data. The character embedding size is set to 30, and the layer number of Bi-LSTM is set to one. We train the recognizer with the learning rate as 0.001 for five iterations. We choose five paths with the highest confidence score in the CRF layer, and treat a token as an API mention if and only if it is predicted so in at least two out of five paths ($Top\_P = 5, K = 2$). For the API linker, we train our library predictor with Transformer [36] for ten iterations with the learning rate of 0.001. The default threshold S for the scoring function is 1.1 unless we specify them with other values.

4.3 Baselines

To the best of our knowledge, there is no existing work focusing on extracting API links from unformatted texts. We compare our method with the following baselines: APIReal is the most relevant work to ours but they mine APIs from StackOverflow posts, and the other two baselines are rule-based.

4.3.1 APIReal. Ye et al. [40] proposed the model named APIReal, which predicted API recognition and linking in a StackOverflow post. APIReal contains two stages similar to ours: a recognizer to extract API mentions and a linker to link API to the repository. In the recognizer, they manually labeled the training data to learn API mentions by feeding human-crafted features into a CRF model. In the linker, they utilized external information, such as the question title, contents in the code block, <code> tags, and *URLs* in a post to predict what an API mention links to.

When implementing this baseline, we "counterfeit" a file crawled from StackOverflow in the same input format, where each line is a sentence from our test set. In this way, APIReal will treat our file as a post from StackOverflow and continuously processes them. Moreover, as the database of APIReal includes three of five libraries comparing to ours (i.e., Pandas, Numpy, Matplotlib), we compute Precision, Recall, and F1 scores on the three libraries.

4.3.2 RuleBase-Pure. We also include a pure Rule-based approach as the baseline. Specifically, we check whether each token in the sentence is the same as an entry in the API repository. This baseline provides us with insights into the quality of written API mentions in StackOverflow.

4.3.3 RuleBase-Knowl. We also include a Rule-based approach with prior knowledge as a baseline. Here, prior knowledge refers to the common writing behaviors for API mentions in StackOverflow. Specifically, we replace *"np"* with *"numpy"*, *"pd"* with *"pandas"*, *"tf"* with *"tensorflow"* for each token, respectively.

4.4 Evaluation Metrics

For fair comparison, we use *Precision*, *Recall*, and *F1* scores to evaluate ARCLIN's performance in our test set, which is also used by all previous works [2, 7, 40]. Specifically, precision means what percentage of API linking predictions are correct, recall means what percentage of the real API mentions are covered, and F1 is the harmonic mean of precision and recall.

5 EXPERIMENTAL RESULTS

In this section, we discuss the performance of ARCLIN model by diving into three research questions from Section 5.1 to Section 5.3:

(1) How effective is ARCLIN? We compare ARCLIN to three baselines in the proposed test set. The result shows that ARCLIN outperforms baselines by large margins, even though it is free from any labor-intensive annotations and handcrafted rules.

(2) How effective are the components of ARCLIN? The devised framework is made up of an API recognizer and an API linker. The latter one includes a library predictor and a scoring function balance between the lexical similarity and entity similarity. To evaluate the contribution of each component, we discard each element at one time and implement the remaining part in our test set. Details of analysis are provided along with the experiment results.

(3) What is the generalization ability of ARCLIN? Considering the large number of libraries in the real world, we are interested in how ARCLIN performs in mining APIs inside an unseen library. To explore its generalization ability, we train the model in one library and test it in another library.

5.1 RQ1: How effective is ARCLIN?

ARCLIN aims to automatically extract API mentions from free text sentences and link them to an entry in the repository. Thus, to prove its effectiveness, we evaluate ARCLIN in sentences selected from StackOverflow posts. We feed test sentences into the ARCLIN model and examine whether it could mine correct APIs.

[5] Word2Vec and FastText models converge before ten epochs.

Table 4: Experimental Results.

Approach	Precision	Recall	F1
RuleBase-Pure	1.00	0.070	0.131
RuleBase-Knowl	1.00	0.314	0.478
APIReal	0.787	0.604	0.683
- w/o rules	0.823	0.477	0.599
ARCLIN Ensemble	0.784	0.742	**0.762**
- PyTorch	0.861	0.801	0.830
- Tensorflow	0.576	0.840	0.683
- Pandas	0.717	0.778	0.746
- Numpy	0.865	0.741	0.798
- Matplotlib	0.825	0.762	0.792

To answer this question, we compare ARCLIN with a current state-of-the-art baseline named APIReal [40], a purely rule-based approach RuleBased-Pure, and a rule-based approach incorporating prior knowledge named RuleBased-Knowl. Experimental results are shown in Table 4. Apart from the overall performance of ARCLIN in our whole test set (in ARCLIN Ensemble), we also examine the performance in every single library, shown in the following five rows. The results indicate our ARCLIN significantly outperforms all other baselines. From the results, we see that our ARCLIN model can achieve 78.41%, 74.19%, and 76.24% in precision, recall, and F1 score, respectively.

It is worthy to notice that RuleBase-Pure only retrieves 6.99% of all API mentions, reflecting that developers rarely write the entire qualified name when they mention some APIs. This is also one of the motivations of this work. The RuleBase-Knowl model provides better performance with the help of prior knowledge. However, if we want to extend the model to a large number of libraries, it is implausible for researchers to enumerate all possible abbreviations for each library. Although the model gives an acceptable performance, it can hardly be used extensively. Another baseline APIReal reaches the F1 score of 0.683, which is lower than the performance in their dataset. We attribute the unfavorable performance to several reasons: (1) The constraints of handcrafted patterns in resolving customized variables. As illustrating in the second morphological challenge, the unprofessional developers usually write down API mentions with customized variables (*a.reshape*) or aliases (*np.reshape*). Since APIReal leverages a collection of pre-defined rules to solve the problem (e.g., *np* for *numpy*), the customized variables or uncommon aliases outside the scope lead to mistakes. The impact of such handcrafted rules is quantitated in the w/o rule line in Table 4. (2) Difficulty in mining APIs in free texts. APIReal leverages <code> tags in its recognizer; thus, once someone writes down API mentions in such tags, APIReals can easily extract them. But our dataset does not contain such signs to help the recognizer find out API mentions. (3) Insufficient information. APIReal utilizes information from source StackOverflow posts, such as *URLs*, question titles, and code snippets. However, mining APIs from sentences in our task requires the model to capture a richer semantic meaning.

ARCLIN reaches the highest performance among the three baselines. We conclude the reasons as follows: (1) ARCLIN owns the recognizer that keeps all possible API mentions by selecting the top five paths in CRF and conduct voting. In this way, ARCLIN will not miss too many API mentions; (2) ARCLIN's library indicator provides scope for library selection, preventing it from linking to the entry from wrong libraries; (3) ARCLIN's scoring function balances the lexical similarity and spelling similarity, so small variations of an API's name will not affect its final prediction.

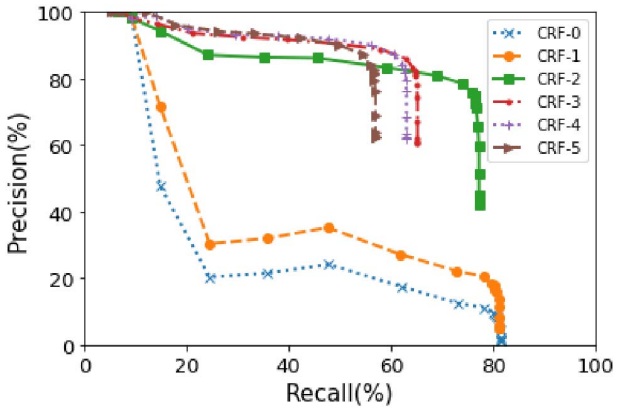

Figure 3: P-R curve of different hyper-parameters K and S.

Table 5: Effectiveness of components in ARCLIN.

	Precision	Recall	F1
ARCLIN (ensemble)	0.784	0.742	0.762
- w/o recognizer	0.112	0.783	0.195
- w/o lib_pred	0.649	0.715	0.680
- w/o lexical_sim	0.814	0.439	0.570
- w/o entity_sim	0.645	0.719	0.680

In addition to the good performance, another advantage of ARCLIN is its flexibility. Figure 3 provides a precision-recall curve to show how the performance is affected by the hyperparameters K and S. Each curve *CRF-K* in the figure represents a token will be considered as an API mention if $K(0 \leq K \leq 5)$ out of five paths predict it so. Each point in a curve is ARCLIN's performance under a similarity threshold $S(0 \leq S \leq 2)$. A higher-scoring threshold means a matched <mention, entry> requires a higher similarity. Generally, a higher precision occurs simultaneously with a lower recall rate. ARCLIN is able to achieve 100% precision under a low recall rate. Therefore, we can customize the threshold under different scenarios. The figure also shows that we cannot achieve 100% recall even the precision gets down to zero. We ascribe the situation into the following reason: Compared with character-disorder, word-disorder is too complex for ARCLIN to deal with. For example, when *torch.nn.BCEWithLogitsLoss* is written as *BCELosswithlogits*, even if ARCLIN narrows down candidates into the correct library, it is hard for ARCLIN to conduct API linking with each other. To conclude, after evaluating ARCLIN in our test set the experiment result shows that it outperforms baselines by large margins, even though it is free from any labor-intensive annotations and handcrafted rules.

Table 6: The generalization ability of ARCLIN. P, R and F1 refers to precision, recall and F1 score respectively.

Training	Testing														
	PyTorch			Tensorflow			Pandas			Numpy			Matplotlib		
	P	R	F1	P	R	F1	P	R	F1	P	R	F1	P	R	F1
PyTorch	-	-	-	0.289	0.753	0.418	0.455	0.650	0.535	0.472	0.741	0.576	0.852	0.657	0.742
Tensorflow	0.427	0.710	0.533	-	-	-	0.420	0.701	0.526	0.422	0.768	0.544	0.738	0.752	0.745
Pandas	0.472	0.634	0.541	0.284	0.753	0.412	-	-	-	0.469	0.741	0.574	0.833	0.667	0.741
Numpy	0.422	0.664	0.516	0.222	0.803	0.348	0.365	0.727	0.486	-	-	-	0.817	0.724	0.768
Matplotlib	0.449	0.641	0.528	0.268	0.765	0.397	0.415	0.684	0.516	0.454	0.741	0.563	-	-	-

5.2 RQ2: How effective are the components of ARCLIN?

ARCLIN is comprised of an *API recognizer* and an API linker with a *library predictor* and a scoring function balance between the *lexical similarity* and *entity similarity*. To investigate the contribution of each module, we discard each component at a time, implement the model in our test set, and analyze its performance. Experimental results of this ablation study are shown in Table 5, where each row below ARCLIN (ensemble) represents the result of a missing component. In the w/o lexical_sim and w/o entity_sim setting, we set the scoring threshold to 0.6. Generally, the missing module negatively affects the model's performance more or less. We will discuss the effects in the following paragraphs respectively.

5.2.1 NO Recognizer. In this setting, the model tries to link an entry in the API repository for each token. The precision performance is dramatically decreasing because the majority of tokens in a sentence are not API mentions, but they can still be linked to an API in the repository because of their high similarity. For instance, the common word *where* has a high similarity with the API *numpy.where()* because both of them contain "where" within the token, but it is not an API mention. ARCLIN made lots of such mistakes, causing low precision.

5.2.2 NO Library Predictor. In this setting, the model tries to generate candidates from all five libraries, neglecting the sentence context information. The failure occurs when different libraries have a method with similar names. For instance, PyTorch has the method *torch.stack()* while Numpy also contains the method *numpy.stack()*, if a developer only writes *"stack"* as the API mention, the model cannot disambiguate the token.

5.2.3 NO Lexical Similarity. Without the lexical similarity, the scoring function fully relies on the entity similarity. A <mention, entry> pair will be linked if and only if some entities within them are exactly the same. This approach provides a high precision rate, since it is similar to an advanced rule-based algorithm. However, it cannot deal with spelling errors. For example, *np.zeros()* will be linked with *numpy.zeros()* because both of them has the entity *"zeros"*, but *numpy.zeros()* cannot matched with *np.zero()*, even if the API mention contains only one missing character.

5.2.4 NO Entity Similarity. In this case, the lexical similarity is determinative to the scoring function. This function works fine in most cases, but falling short when an API mention refers to a long API name. For instance, the API mention *tf.layers.batch_normalization* has a higher similarity score with *tf.keras.layers.BatchNormalization()* rather than *tf.compat.v1.layers.batch_normalization()*. From a lexical perspective, *batch_normalization* is not far away from *BatchNormalization*, so the final scoring function will easily be affected by other factors (i.e., missing module name in this example).

5.3 RQ3: What is the generalization ability of ARCLIN?

Considering the large number of libraries even for one programming language, we are interested in the generalization ability of ARCLIN. A promising API mining model should have the ability to mine APIs without training on the library-specific corpus.

To answer the question, we train the recognizer and linker in one library corpus, then the model attempts to recognize and identify APIs of another library in our test set. In this setting, the model never sees the new library before, so the library predictor is removed from ARCLIN.

Table 6 shows the generalization ability of each pair of libraries. The experimental results show ARCLIN gains the generalization ability to some extent. The experiments further indicates that the transferred model evaluated with Matplotlib achieves a higher performance. For instance, the model that has been trained from Numpy, is able to correctly recognize 81.7% Matplotlib APIs, according to the Table 6. We ascribe the reason to the distinctiveness of API's name in Matplotlib. Specifically, API names in Matplotlib (e.g., *matplotlib.pyplot.pcolormesh()* are rather different from APIs in scientific computing libraries (e.g., *numpy.zeros()* or *torch.zeros()*), so the model is free from mistakenly linking to APIs in other libraries.

Generally, the transferred models contain a better recall rate rather than precision, and we discuss the reason as follows. Without library predictor, ARCLIN may link API mentions to the wrong library if they contain similar method names. For example, given a sentence "I have trouble with concatenating a list of tensors using PyTorch's stack." where *"stack"* here is labeled as *torch.stack()* in ground truth during the testing phase. If we train the model in Pandas and evaluate its generalization ability in Numpy library, ARCLIN will link *"stack"* to the API *numpy.stack()*. In summary, the experiment results demonstrate the effectiveness and robustness of generalization ability. Such library-transferred experiment mimics the real-world scenario of applying ARCLIN to mine APIs from unseen libraries.

Table 7: Case study.

Approach	Rule-K	APIReal		ARCLIN	
API Mention	-	Recog	Link	Recog	Link
figure.add_suplot	✗	✗	✗	✓	✓
ax.set_major_locator	✗	✓	✗	✓	✓
ticker.MultipleLocator	✗	✓	✓	✓	✓
plt.show()	✓	✓	✓	✓	✓

6 CASE STUDY

In this section, we dive into three cases to specify why ARCLIN outperforms APIReal and Rule-K (i.e., RuleBase-Knowl), where ground-truth is shown in blue italic font. The experimental results of four API mentions (in blue) are presented in Table 7 with respect to the two phases (i.e., API Recognizer and API Linker[6]). One API mention is successfully identified and resolved if and only if the "Link" phase gives the correct answer (✓).

- Is there a more convenient alternative to *figure.add_suplot [matplotlib.figure.Figure.add_subplot()]* if I have multiple figures ...
- You may try ticking the major axis using *ax.set_major_locator [matplotlib.axis.Axis.set_major_locator()]* called with *ticker.MultipleLocator() [matplotlib.ticker.MultipleLocator]*.
- If you don't want to export, please uncomment *plt.show() [matplotlib.pyplot.show()]* and remove ...

Compared ARCLIN with Rule-K and APIReal, we categorize the characteristics for three approaches. Firstly, Rule-K can only resolve the API mention with the qualified name or a collection of specific abbreviations, depending on the handcrafted rules. For instance, if we add the common writing behavior that a developer usually calls *matplotlib.pyplot* by its alias *plt*, Rule-K will try to replace the alias with its original name for each token, then find if the new token matches a fully qualified API name in the repository. Secondly, we observe that APIReal is more flexible than rule-based matching algorithm, by uncovering some API mentions by the recognizer (e.g., *ax.set_major_locator* and *ticker.MultipleLocator*), allowing it to address the first common-word ambiguity challenge. Nevertheless, its API Linker is not perfect to resolve the ambiguity introduced by morphological mentions, mainly comes from the customized name, such as *ax* or *ticker*. APIReal detects aliases by handcrafted patterns (e.g., *pd* for *pandas*), thus the alias that is not covered by rules will be inappropriately coped with. Last but not least, the cases demonstrate the effectiveness of ARCLIN. The carefully devised API recognizer enables it to detect API mentions in unformatted text. Besides, the API Linker with entity similarity forces the model to pay attention to the informative entities (e.g., *set_major_locator*), and the lexical similarity allows it to address the misspelling in API mentions. Therefore, ARCLIN can even resolve the *figure.add_suplot* to *matplotlib.figure.Figure.add_subplot()* even if the mention leaves out the letter "b".

[6]API Recognizer is denoted as Recog and API Linker is denoted as Link for space limitation.

7 THREAT TO VALIDITY

In this section, we discuss three potential threats to the validity of ARCLIN and provide our solutions to alleviate these threats. The first one is the potential bias brought by manual annotation of the data. We evaluate ARCLIN the Py-mention dataset, which is annotated by two different annotators. To overcome the human bias and ensure the data quality, we not only employ domain experts instead of crowd-sourcing workers, but also throw away the sentences with uncertainty. Annotation examples and guidelines are provided at first. As a result, the annotators fully understand what they need to do and keep confidence in their annotation.

The second one is the limited recall rate. As shown in Figure 3, the recall cannot achieve 100% regardless of the threshold. In other words, ARCLIN cannot cover all ground-truth labels. We owe this recall limitation to the reasons of observed word disorder in mentions. ARCLIN computes similarity based on lexical-level and entity-level, but it fails in comparing <mention, entry> pairs in word disorder. For example, if we use *BCELosswithlogits* to represent *torch.nn.BCEWithLogitsLoss*, the similarity score from ARCLIN is close to *torch.nn.BCELoss*, therefore, the final output gets perturbation by other factors. To alleviate the issue, an n-gram based similarity can be used to extend our ARCLIN model.

The third threat is style constraint. Currently, we evaluate ARCLIN with five Python libraries and achieve promising performance, but if we migrate the model to other programming languages, the inconsistency of function calling format will introduce this threat. For instance, in C++ language, we use double colon "::" to call a static function or declare the namespace identification. Besides, to call a function in a class, one may use "->" from a pointer or use the node "." from a C++ entity. ARCLIN uses "." to split the API's entire qualified name into a bag of package entities for similarity computation. If we implement ARCLIN in another language (e.g., C++), it is necessary to implement new split marks.

8 RELATED WORKS

API Recognition. If we want to link an API to some other source, the first step is to recognize APIs. In this paper, we use a recognizer to recognize APIs in free text sentences. Dagenais and Robillard [7] adopted partial program analysis (PPA) to parse Java snippets and then extracts code-like terms in informal discussions.The difference between theirs and ours is, our paper targets extracting APIs from natural language sentences, but the above studies were about extracting APIs from code blocks (written in free texts). Bacchelli et al. [2] employed a rule-based approach to extract API mentions from e-mails by designing different regular expressions applicable to different languages. Treude and Robillard [35] suggested different regular expressions for question and body to extract API mentions from StackOverflow posts. Rigby and Robillard [31] used island grammars to identify code elements from free text with the help of compound camel cased terms while ignoring the common-word ambiguity. The most relevant research to us is APIReal [40], but their approach was applicable to recognizing APIs from StackOverflow posts with <code> tags, which was much easier than our setting. Besides, instead of linking such fine granularity APIs, researchers also explored linking between textual documents and code artifacts for maintenance. Some works Antoniol et al. [1], Chen [6], Marcus

and Maletic [23], Marcus et al. [24] used information retrieval (IR) techniques or leverage Latent Semantic Indexing (LSI) to recover traceability links between elements in natural language documentation and source code in software systems. These studies were different from this paper since they performed a coarse granularity linking.

API linking. "Linking" can refer to linking code artifacts to documents, or linking APIs from free-texts to its entire qualified name in the repository. Regarding to linking code artifacts to documents, Bacchelli et al. [2] used two string-match information retrieval techniques (i.e., vector space model and LSI) to link detected APIs from e-mails to source code artifacts. The latter category is what we have done in this paper, the main idea of matching APIs with their entire qualified name is how to conduct the disambiguation. Dagenais and Robillard [7] suggested a set of filtering heuristics to disambiguate the API mentions. Ye et al. [40] disambiguated API mentions in a StackOverflow post by utilizing information in code blocks, question titles, and the location where *URLs* points to. The first paper did not address the common word polysemy, while the second research mitigated the morphological challenge by labor-intensive rules, which was different from ours.

Mining Technical Forums. Nowadays, many researchers devote themselves to mining knowledge from technical forums (e.g., StackOverflow) to facilitate developers in their programming issues. For example, a popular scenario is API recommendation [18, 26, 37], these papers suggested a list of API classes for a natural language query by mining StackOverflow posts. Specifically, given a natural language query, Huang et al. [18] firstly searched the most relevant 50 questions and extracting APIs from posts. Then, it ranked all candidate APIs by considering the query-title similarity and title-APIs similarity. Li et al. [21] proposed another application that explores API caveat in such a technical forum and presented a system to help developers to tackle the problem of negative usage of APIs. It is noticed that many works studied in StackOverflow talks about APIs, our work serves as a foundation of this work for facilitating them to recognize and identify the APIs without the entire qualified name.

9 CONCLUSION

In this paper, we propose a novel framework ARCLIN for recognizing API mentions from free text and linking to an API repository. ARCLIN is composed of two components, an API recognizer and an API linker. The API recognizer extracts API mentions from free texts and the API linker disambiguates the API mentions by a library predictor to address reference ambiguity, and a scoring function incorporating lexical similarity and entity similarity. After training the model in an unlabeled StackOverflow corpus, we implement ARCLIN in a human-annotated dataset named Py-mention, the experimental results demonstrate that it significantly outperforms all baselines. Moreover, the experiment about generalization ability demonstrates that ARCLIN can extract APIs from a new library even though ARCLIN is trained from another libraries.

10 ACKNOWLEDGEMENT

The work was supported by the Guangdong Key Research Program (No. 2020B010165002) and the Research Grants Council of the Hong Kong Special Administrative Region, China (CUHK 14210920).

REFERENCES

[1] Giuliano Antoniol, Gerardo Canfora, Gerardo Casazza, Andrea De Lucia, and Ettore Merlo. 2002. Recovering traceability links between code and documentation. *IEEE transactions on software engineering (TSE)* 28, 10 (2002), 970–983.
[2] Alberto Bacchelli, Michele Lanza, and Romain Robbes. 2010. Linking e-mails and source code artifacts. In *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering-Volume 1 (ICSE)*. 375–384.
[3] Steven Bird, Ewan Klein, and Edward Loper. 2009. *Natural language processing with Python: analyzing text with the natural language toolkit.* " O'Reilly Media, Inc.".
[4] Piotr Bojanowski, Edouard Grave, Armand Joulin, and Tomas Mikolov. 2017. Enriching word vectors with subword information. *Transactions of the Association for Computational Linguistics (TACL)* 5 (2017), 135–146.
[5] Chunyang Chen, Zhenchang Xing, and Ximing Wang. 2017. Unsupervised software-specific morphological forms inference from informal discussions. In *Proceedings of the 39th IEEE/ACM International Conference on Software Engineering (ICSE)*. IEEE, 450–461.
[6] Xiaofan Chen. 2010. Extraction and visualization of traceability relationships between documents and source code. In *Proceedings of the IEEE/ACM international conference on Automated software engineering (ASE)*. 505–510.
[7] Barthélémy Dagenais and Martin P Robillard. 2012. Recovering traceability links between an API and its learning resources. In *Proceedings of the 34th International Conference on Software Engineering (ICSE)*. IEEE, 47–57.
[8] Marie-Catherine De Marneffe, Bill MacCartney, Christopher D Manning, et al. 2006. Generating typed dependency parses from phrase structure parses.. In *Proceedings of the Eleventh International Conference on Language Resources and Evaluation (LREC)*, Vol. 6. 449–454.
[9] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. 2019. BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding. In *Proceedings of the 2019 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, (NAACL-HLT)*. Association for Computational Linguistics, 4171–4186.
[10] Matplotlib Documentation. Retrieved in 2021. https://matplotlib.org/stable/contents.html
[11] Numpy Documentation. Retrieved in 2021. https://numpy.org/devdocs/reference/index.html
[12] PyTorch Documentation. Retrieved in 2021. https://pytorch.org/docs/stable/index.html
[13] Pandas Documentation. Retrieved in 2021. https://pandas.pydata.org/docs/reference/index.html#api
[14] Tensorflow Documentation. Retrieved in 2021. https://www.tensorflow.org/api_docs/python/tf
[15] Wei Fu and Tim Menzies. 2017. Easy over hard: A case study on deep learning. In *Proceedings of the 11th the ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE)*. 49–60.
[16] Alex Graves, Abdel-rahman Mohamed, and Geoffrey Hinton. 2013. Speech recognition with deep recurrent neural networks. In *International Conference on Acoustics, Speech and Signal Processing, Vancouver, BC, Canada, May 26-31, 2013*. IEEE, 6645–6649. https://doi.org/10.1109/ICASSP.2013.6638947
[17] Matthew Honnibal, Ines Montani, Sofie Van Landeghem, and Adriane Boyd. 2020. *spaCy: Industrial-strength Natural Language Processing in Python.*
[18] Qiao Huang, Xin Xia, Zhenchang Xing, David Lo, and Xinyu Wang. 2018. API method recommendation without worrying about the task-API knowledge gap. In *Proceedings of the 33rd IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 293–304.
[19] Paul Jaccard. 1912. The distribution of the flora in the alpine zone. 1. *New phytologist* 11, 2 (1912), 37–50.
[20] John D. Lafferty, Andrew McCallum, and Fernando C. N. Pereira. 2001. Conditional Random Fields: Probabilistic Models for Segmenting and Labeling Sequence Data. In *Proceedings of the Eighteenth International Conference on Machine Learning (ICML)*. Morgan Kaufmann, 282–289.
[21] Jing Li, Aixin Sun, Zhenchang Xing, and Lei Han. 2018. API Caveat Explorer–Surfacing Negative Usages from Practice: An API-oriented Interactive Exploratory Search System for Programmers. In *Proceedings of the 41st International ACM SIGIR Conference on Research & Development in Information Retrieval (SIGIR)*. 1293–1296.
[22] Xuezhe Ma and Eduard Hovy. 2016. End-to-end Sequence Labeling via Bi-directional LSTM-CNNs-CRF. In *Proceedings of the 54th Annual Meeting of the Association for Computational Linguistics (ACL)*. Association for Computational Linguistics, 1064–1074.
[23] Andrian Marcus and Jonathan I Maletic. 2003. Recovering documentation-to-source-code traceability links using latent semantic indexing. In *Proceedings of the 25th International Conference on Software Engineering (ICSE)*. IEEE, 125–135.
[24] Andrian Marcus, Jonathan I Maletic, and Andrey Sergeyev. 2005. Recovery of traceability links between software documentation and source code. *International Journal of Software Engineering and Knowledge Engineering* 15, 05 (2005), 811–836.

[25] Tomas Mikolov, Ilya Sutskever, Kai Chen, Greg Corrado, and Jeffrey Dean. 2013. Distributed representations of words and phrases and their compositionality. *arXiv preprint arXiv:1310.4546* (2013).

[26] Mohammad Masudur Rahman, Chanchal K Roy, and David Lo. 2016. Rack: Automatic api recommendation using crowdsourced knowledge. In *Proceedings of the 23rd IEEE International Conference on Software Analysis, Evolution, and Reengineering (SANER)*, Vol. 1. IEEE, 349–359.

[27] Lev Ratinov and Dan Roth. 2009. Design challenges and misconceptions in named entity recognition. In *Proceedings of the 13th Conference on Computational Natural Language Learning (CoNLL)*. 147–155.

[28] Radim Řehůřek and Petr Sojka. 2010. Software Framework for Topic Modelling with Large Corpora. In *Proceedings of the LREC 2010 Workshop on New Challenges for NLP Frameworks*. ELRA, 45–50.

[29] Xiaoxue Ren, Jiamou Sun, Zhenchang Xing, Xin Xia, and Jianling Sun. 2020. Demystify official API usage directives with crowdsourced API misuse scenarios, erroneous code examples and patches. In *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering (ICSE)*. 925–936.

[30] Xiaoxue Ren, Xinyuan Ye, Zhenchang Xing, Xin Xia, Xiwei Xu, Liming Zhu, and Jianling Sun. 2020. API-Misuse Detection Driven by Fine-Grained API-Constraint Knowledge Graph. In *2020 35th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 461–472.

[31] Peter C Rigby and Martin P Robillard. 2013. Discovering essential code elements in informal documentation. In *Proceedings of the 35th International Conference on Software Engineering (ICSE)*. IEEE, 832–841.

[32] Martin Sundermeyer, Ralf Schlüter, and Hermann Ney. 2012. LSTM neural networks for language modeling. In *Annual Conference of the International Speech Communication Association, Portland, Oregon, USA, September 9-13, 2012*. ISCA, 194–197. http://www.isca-speech.org/archive/interspeech_2012/i12_0194.html

[33] Jeniya Tabassum, Mounica Maddela, Wei Xu, and Alan Ritter. 2020. Code and Named Entity Recognition in StackOverflow. In *Proceedings of the 58th Annual Meeting of the Association for Computational Linguistics (ACL)*. Association for Computational Linguistics, 4913–4926.

[34] Erik F. Tjong Kim Sang and Fien De Meulder. 2003. Introduction to the CoNLL-2003 Shared Task: Language-Independent Named Entity Recognition. In *Proceedings of the 7th Conference on Natural Language Learning at HLT-NAACL*. 142–147.

[35] Christoph Treude and Martin P Robillard. 2016. Augmenting api documentation with insights from stack overflow. In *Proceedings of the 38th IEEE/ACM International Conference on Software Engineering (ICSE)*. IEEE, 392–403.

[36] Thomas Wolf, Lysandre Debut, Victor Sanh, Julien Chaumond, Clement Delangue, Anthony Moi, Pierric Cistac, Tim Rault, Rémi Louf, Morgan Funtowicz, Joe Davison, Sam Shleifer, Patrick von Platen, Clara Ma, Yacine Jernite, Julien Plu, Canwen Xu, Teven Le Scao, Sylvain Gugger, Mariama Drame, Quentin Lhoest, and Alexander M. Rush. 2020. Transformers: State-of-the-Art Natural Language Processing. In *Proceedings of the 2020 Conference on Empirical Methods in Natural Language Processing: System Demonstrations*. Association for Computational Linguistics, 38–45.

[37] Wenkai Xie, Xin Peng, Mingwei Liu, Christoph Treude, Zhenchang Xing, Xiaoxin Zhang, and Wenyun Zhao. 2020. API method recommendation via explicit matching of functionality verb phrases. In *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 1015–1026.

[38] Bowen Xu, Deheng Ye, Zhenchang Xing, Xin Xia, Guibin Chen, and Shanping Li. 2016. Predicting semantically linkable knowledge in developer online forums via convolutional neural network. In *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 51–62.

[39] Jie Yang and Yue Zhang. 2018. NCRF++: An Open-source Neural Sequence Labeling Toolkit. In *Proceedings of the 56th Annual Meeting of the Association for Computational Linguistics (ACL)*.

[40] Deheng Ye, Lingfeng Bao, Zhenchang Xing, and Shang-Wei Lin. 2018. APIReal: an API recognition and linking approach for online developer forums. *Empirical Software Engineering (ESE)* 23, 6 (2018), 3129–3160.

[41] Deheng Ye, Zhenchang Xing, Chee Yong Foo, Zi Qun Ang, Jing Li, and Nachiket Kapre. 2016. Software-specific named entity recognition in software engineering social content. In *Proceedings of the 23rd IEEE International Conference on Software Analysis, Evolution, and Reengineering (SANER)*, Vol. 1. IEEE, 90–101.

[42] Deheng Ye, Zhenchang Xing, Chee Yong Foo, Jing Li, and Nachiket Kapre. 2016. Learning to extract api mentions from informal natural language discussions. In *Proceedings of the 32nd IEEE International Conference on Software Maintenance and Evolution (ICSME)*. IEEE, 389–399.

[43] Haoxiang Zhang, Shaowei Wang, Tse-Hsun Chen, and Ahmed E Hassan. 2019. Reading answers on stack overflow: Not enough! *IEEE Transactions on Software Engineering (TSE)* (2019).