JARAD: An Approach for Java API Mention Recognition and Disambiguation in Stack Overflow

Qingmi Liang, Yi Jin, Qi Xie, Li Kuang(✉), and Yu Sheng

School of Computer Science, Central South University, Changsha 410018, China
{qmliang,8209200329,8209190322,kuangli,shengyu}@csu.edu.cn

Abstract. Invoking APIs is a common way to improve the efficiency of software development. Developers often discuss various problems encountered or share the experience of using the API in communities, like Stack Overflow and GitHub. To avoid the duplicate discussion of issues and support downstream tasks such as API recommendation and API Mining, it is necessary to recognize APIs mentioned in these communities and link them to the fully qualified name. This work is often referred to as the task of API mention recognition and disambiguation in informal texts, which is the main focus of our paper. We start from Java posts in Stack Overflow and analyze the proportion of the posts that involve discussion on API (API Post for short), with short names or fully qualified names, and the characteristics of API Post. We also extract the APIs associated with more than 30,000 posts in Stack Overflow, and automatically establish $< post, APIs >$ pairs to construct the dataset JAPD. Finally, we propose a novel approach JARAD to infer the associated APIs in a post. In our approach, we first use BiLSTM and CRF to fuse context information in text and code snippets to obtain a set of associated API candidates. The candidate API is then scored by the frequency of the API type appearing in the post to infer API's fully qualified name. Our evaluation experiments demonstrate that JARAD achieves 71.58%, 76.84% and 74.12% on Precision, Recall and F1 respectively.

Keywords: Java API · Mention Recognition · API Disambiguation · Research Analysis · Dataset · Stack Overflow

1 Introduction

Invoking the Application Programming Interface (API) is an effective means for developers to reuse code. The official API documentation is an effective means to guide developers to call APIs correctly. However, due to the limited information and scattered knowledge in the official documentation, developers have to discuss various problems that they encountered in the process of calling APIs in the communities, like Stack Overflow and GitHub. They always refer to APIs in discussions by their simple names and aliases, which makes it more difficult for

H. Gao et al. (Eds.): CollaborateCom 2023, LNICST 561, pp. 265–283, 2024.
https://doi.org/10.1007/978-3-031-54521-4_15

developers to search API-related knowledge in the communities. It not only leads to knowledge waste but also brings difficulties to some downstream tasks such as API recommendation [15,16,33] and API Mining [15,16,18,19].

To know exactly which APIs are referred to in the discussion, extensive research was performed to recognize APIs mentioned in discussions, and infer APIs' fully qualified names. In the early days, researchers used rule-based methods [3,17]. For example, Christoph Treude [3] solved the API mention recognition task by summarizing regular expression rules after observing a large number of sentences. Later, some methods based on machine learning [1,2,10,13,14] also became popular in this field. ARCLIN [2] recognizes API mentions of text fragments in posts based on neural networks, and infers the fully qualified name of the associated API using the similarity comparison.

However, existing methods for API mention recognition and disambiguation field still suffer from the following deficiencies: (1) There is no research to analyze the proportion of developers' discussions about APIs and what characteristics exist in these discussions, resulting in a lack of clear understanding of this work; (2) There is no available, sufficient, and general dataset to support the research. Researchers have to annotate the dataset manually [2,5], which undoubtedly brings tedious and repetitive work to researchers; (3) Some existing research methods have certain limitations: some methods [2,10] only exploit the information of the text fragment, ignoring code snippets in the discussion. However the discussants will also attach relevant codes, which are also an effective basis to help reason about the exact API. Some methods are only aimed to a few specific libraries [1,2,5], rather than general-purpose development languages.

To solve the above dilemmas, this paper conducts a series of researches about the posts in Stack Overflow. First, we filtered out posts not related to Java, investigated the proportion of API Post (the posts that involve discussions on API, API Post for short) in Java posts and with fully qualified names, and analyzed related features for API Post. Next, we automatically labeled more than 30,000 entries based on the APIs' fully qualified names and their links in the official Java API documentation. We automatically pointed out the fully qualified names of the APIs involved in the posts establishing $< post, APIs >$ pairs and constructed a general dataset JAPD (**J**ava **A**PI **P**ost **D**ataset). Finally, we proposed an approach called JARAD to decompose the task into **J**ava **A**PI **R**ecognition **A**nd **D**isambiguation. We first used Bidirectional Long Short-Term Memory (BiLSTM) and Conditional Random Fields (CRF) fusing contextual information to identify API mentions in text and code snippets, and initially obtained the set of API candidates. We then scored the candidate APIs using the frequency of the API type in the post and obtain the fully qualified names of the APIs associated with the post.

To verify the effectiveness of our method, we evaluated JARAD on the proposed JAPD. We tested the effect of two components of JARAD, i.e. API mention recognition and disambiguation. And JARAD achieved a Precision of 71.58%, Recall of 76.84%, and F1 measure of 74.12%. Overall, the main contributions of this paper are as follows:

1. To our best knowledge, we are the first to conduct a research analysis on Java API Post in Stack Overflow. We observe the relevant characteristics of API Post, which provides a basis to construct the dataset and propose the JARAD.
2. We provide a usable dataset called JAPD for the field of API mention recognition and disambiguation. We used a text-based approach and labeled 35,825 records automatically. This effectively relieves the pressure on researchers to label data manually.
3. We propose a novel approach called JARAD to automatically infer the fully qualified names of Java APIs involved in a post, taking into account both text and code snippets in the discussion. JARAD performs better on Recall than the state-of-the-art model ARCLIN based on the Python dataset.

The data related to JARAD is available at link[1]. The rest of the paper is organized as follows. The Sect. 2 is the research analysis of this paper. Section 3 presents our dataset for API recognition and disambiguation in Stack overflow for Java. Section 4 and 5 present our approach and results, respectively, and discuss the strengths and weaknesses of this paper. Section 6 introduces the related work. Finally, we concluded the paper in Sect. 7.

2 Research Analysis

2.1 Research Problem

To study the API recognition and disambiguation tasks of code and text in Stack Overflow posts more clearly, we conduct research on posts with the following questions:

1. What is the percentage of API Post out of all Java-related posts?
2. In API Post, what is the percentage of APIs that appear with their full name?
3. What are the key features for associating API in the API Post?

Investigating the first two questions helps us understand the need for research in this field. The results of the third problem help us propose the better approach called JARAD to solve API recognition and disambiguation tasks. It should be noted that we define Java API Post as the post with Java API provided by the official Java JDK or a third-party dependent package, but not user-defined method. Furthermore, we define API Fully Qualified Name Post as the post with API appearing in the form of a fully qualified name.

2.2 Research Observation

The data for the research in this paper comes from the official dataset provided by Stack Overflow[2]. The data update time is March 7, 2022 and the total number

[1] https://anonymous.4open.science/r/JARAD-EDAE.
[2] https://archive.org/details/stackexchange.

of posts reaches 55,513,870. We first observed the tag search results[3] with 'Java' as the search keyword, and took 11 tags as the investigation targets: $< java >$, $< java - 8 >$, $< javadoc >$, $< java - stream >$, $< java.util.scanner >$, $< java - io >$, $< java - time >$, $< java.util.concurrent >$, $< java - 2d >$, $< javax.imageio >$, and $< java - threads >$. The total number of posts related to these tags is 1,879,900. Their proportions are shown in the 'Proportion' column of Table 1. It is not practical to check these posts manually. Therefore, we adopt the statistical sampling [20] and obtained the posts to be investigated. In order to ensure a certain degree of confidence in the error range of the estimated accuracy, we set the number of samples to 384 based on previous research experience [21,22]. However, due to the large proportion of posts with $< Java >$ tags, we set the sampling quantity, as shown in the 'Sampling Quantity' column of Table 1 to control the number of posts for each tag within a reasonable range.

Table 1. Related data of empirical study.

Tags	PostNum	Proportion	Sampling Quantity	API Post Quantity	API Fully Qualified Name Post
java	1830876	97.39%	104	63	10
java-8	22108	1.18%	100	84	11
javadoc	2835	0.15%	100	14	4
java-stream	10354	0.55%	10	9	0
java.util.scanner	6134	0.33%	10	10	7
java-io	1696	0.09%	10	10	5
java-time	1513	0.08%	10	6	2
java.util.concurrent	1328	0.07%	10	8	5
java-2d	1071	0.06%	10	9	0
javax.imageio	1054	0.06%	10	10	2
java-threads	931	0.05%	10	10	2
Total	1879900	1	384	233	48

To add credibility to the survey, we used majority voting to answer the three questions above for each post. Two researchers investigated the posts and answered the questions above. Their observations are carried out independently without interfering with each other. They both have more than three years of Java development experience. If they have different or uncertain answers to a question, the third researcher who has more than five years of Java development experience is asked to answer the question.

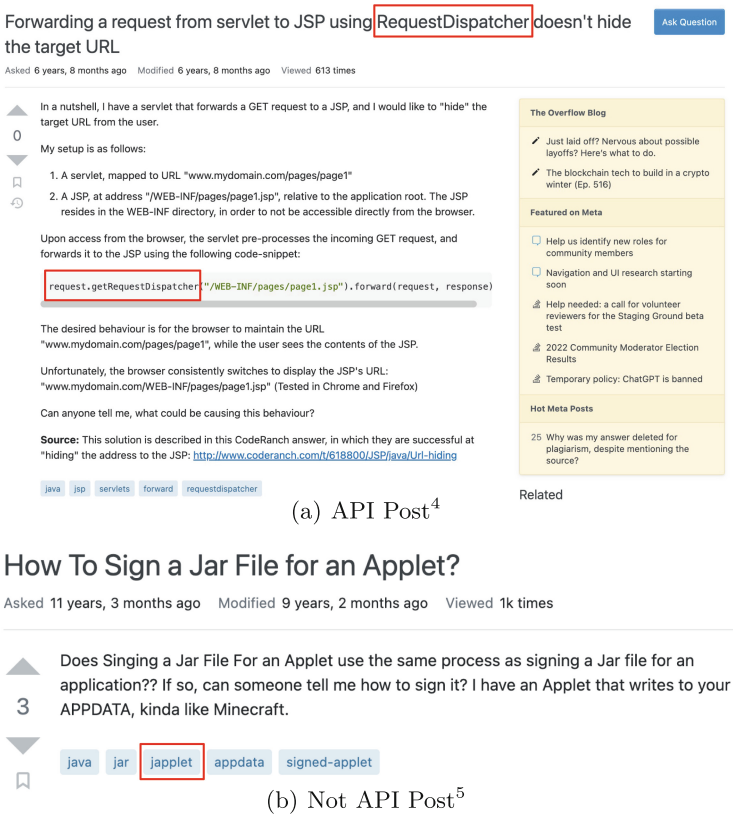
2.3 Research Result

We use the statistic Cohen's Kappa to evaluate the results. If Cohen's Kappa greater than 0.8, there is firm agreement between the results of the two individuals. In our research, Cohen's Kappa of our two researchers achieved 0.91.

[3] https://stackoverflow.com/tags.

Through the investigation of posts by three researchers, we came to the following conclusions.

API Post Make up 61% of all Java Post. There are 233 API Post of 384 Java Post. The proportion of the API Post tagged with $< Java >$ or $< Javadoc >$ is the lowest among the 11 tags we selected. The proportion of API Post tagged with $< java - stream >$, $< java.util.scanner >$, $< java - io >$, $< java{-}time >$, $< java.util.concurrent >$, $< java{-}2d >$, $< javax.imageio >$, and $< java{-}threads >$ achieve 90%. It is not difficult to find that these tags with high proportion are package names related to the Java development language. Figure 1 shows two examples of an API Post and a non-API Post. As shown in Fig. 1(b), it cannot be inferred that the post is discussing the API even if the class tag 'japplet' of Java exists in the post. In other words, the tag can be used as an enhanced condition for judging whether there is an API mention, but it cannot be used as a sufficient condition for API Post or to filter API Post.

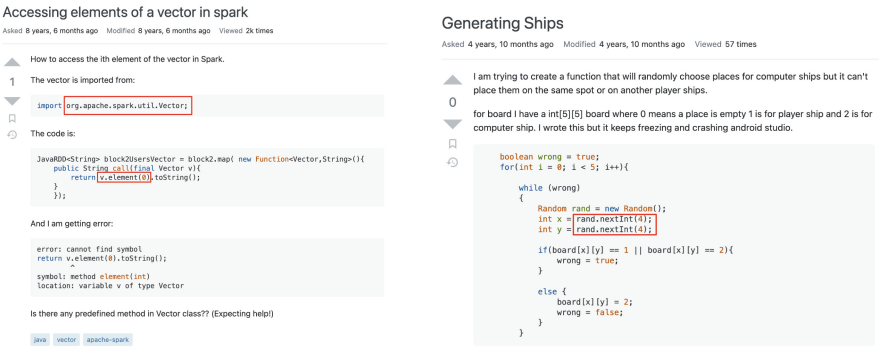


(a) API Post[4]

(b) Not API Post[5]

Fig. 1. Two examples of an API Post and a non-API Post (https://stackoverflow.com/questions/36255963, https://stackoverflow.com/questions/7191411)

In API Post, the proportion of APIs Appearing as Full Names is 21%. The API in the API Fully Qualified Name Post does not necessarily appear directly in the form of the fully qualified name Package.Class.Method. As long as the text or code in the discussion can indicate that it belongs to the package and class of the called API. Similarly, if this condition is not met, the API is considered to appear in the form of a short name, as shown in Fig. 2(b). We found that among the 233 API posts, the API in 185 API posts appeared in the form of a short name, accounting for 79%. In other words, the exact APIs are unknown in 79% of API Post. This is a waste of API knowledge and adds certain degree of difficulty to the research work of API Mining. It also shows the necessity for informal text API mention recognition and disambiguation work.

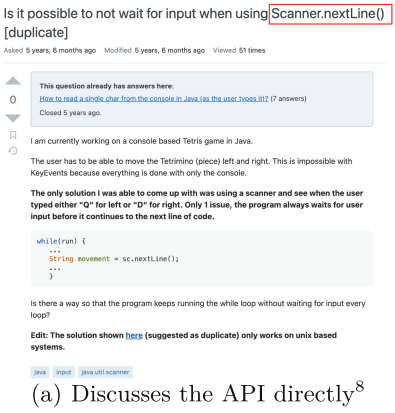


(a) An example of full name API in API Post[6]

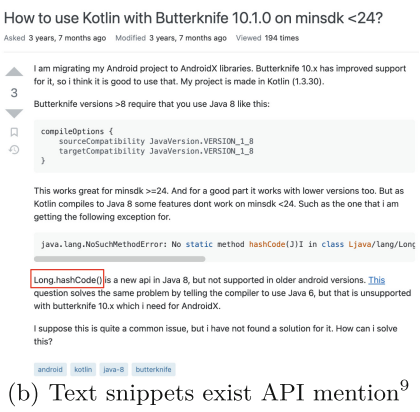
(b) An example of short name API in API Post[7]

Fig. 2. Two examples of API Post (https://stackoverflow.com/questions/23778787, https://stackoverflow.com/questions/48596384)

Code Snippets Have a Higher Percentage of API Mentions than Text Snippets in Posts. Through the observation, we mainly divide the API mentions in posts into the following four types as shown in Fig. 3: (1) The post directly discusses the API (shown in Fig. 3(a)). In this case, API mentions may appear in three places in a post: the title, text in the body, and code in the body. (2) API is mentioned in the text snippet of the post. The discussion does not directly mention the API, but will relate the API in other details. (3) There are API mentions in code snippets of the post, which is similar to the second case. (4) Other situations that do not belong to the above three, such as API mentions in the image. We counted the API mentions of 233 API Post and results are shown in Table 2. Among the 233 API Post, the proportion of posts directly discussing API-related issues is 15.38%, the proportion of API mentions in text snippets is only 17.31%, and the proportion of API mentions in code reaches 66.92%. The probability of an API mention appearing in a picture is extremely low. It should be noted that the reason why the 'Total' in Table 2 is greater than 233 is that some posts have API mentions in both text and code.

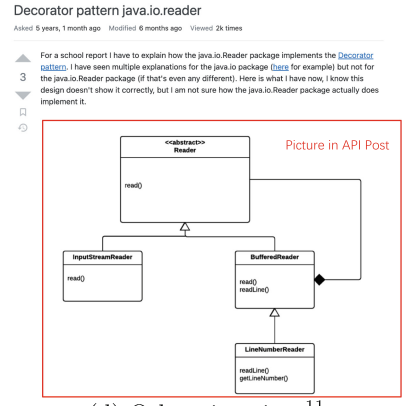


(a) Discusses the API directly[8]



(b) Text snippets exist API mention[9]

(c) Code snippets exist API mention[10]



(d) Other situations[11]

Fig. 3. Examples of API mentions in posts (https://stackoverflow.com/questions/44072821, https://stackoverflow.com/questions/55689197, https://stackoverflow.com/questions/21753769, https://stackoverflow.com/questions/46994892)

Table 2. API Mention statistics of API Posts

API Mention	Quantity	Proportion
Ask related question	40	15.38%
Text	45	17.31%
Code	174	66.92%
Other	1	0.39%
Total	260	1.0

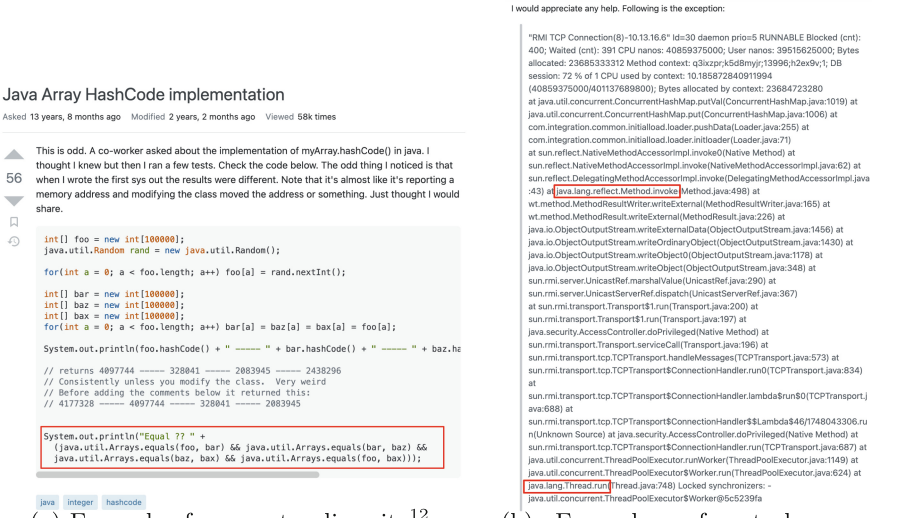
3 Dataset

The lack of available datasets is a non-negligible problem in API mention recognition and disambiguation in informal texts research field, which hinders the advancement of research in this field to a certain extent. Researches [2,5] in recent years have to rely on manual labeling to construct the dataset. With millions of posts, this approach is obviously not realistic. So it is an inevitable trend to construct the dataset automatically. This paper first obtains fully qualified names of all APIs and their links according to the official Java JDK documentation[4], which is called ANAL (**A**PIs' fully qualified **N**ames **A**nd their **L**inks). It is unrealistic to study all APIs and some user-defined APIs usually lack generality and representativeness. We chose the widely used java official API. Then we filter irrelevant data according to the appearance of ANAL in posts on Stack Overflow. After cleaning and optimizing the dataset, 35,825 pieces are obtained and the JAPD (**J**ava **A**PI **P**ost **D**ataset) is constructed. The specific steps to construct the JAPD are as follows:

1. Get APIs' fully qualified names and their links. To ensure data accuracy and increase persuasiveness, we get ANAL and unify the format. We use the requests and the BeautifulSoup library in Python to parse HTML and get the ANAL. We convert API fully qualified name to the standard format of Package.Class.Method (Parameter), such as: java.io.File.equals (java.lang.Object). The parameter is also expressed in the form of a fully qualified name. We unify the API link into the standard format of https://docs.oracle.com/javase/8/docs/api/Package/Class.html#Method-Parameter-, such as: https://docs.oracle.com/javase/8/docs/api/java/io/File.html#equals-java.lang.Object-.

2. Filter API Post by the text-matching approach. We filter API Post from 1,838,095 posts related to 11 tags (see Sect. 2). First, we parse the XML data provided by Stack Overflow to extract major information including Id, Title, Body, and Tags. Next, we use the text-matching approach to filter irrelevant posts according to the ANAL. If there is ANAL for the text snippets(including Title and text in Body) or code snippets in the post, the matched API will be regarded as the associated API of the post. Then we get the $< post, APIs >$ pairs. This approach effectively guarantees the accuracy. When matching the fully qualified name, we ignore the parameter matching because the form of the parameter in the post is diverse as shown in Fig. 4(a). Finally, we delete duplicate pairs, and initially obtained 110,171 $< post, APIs >$ pairs.

[4] https://docs.oracle.com/javase/8/docs/api/.



(a) Example of parameter diversity[12]

(b) Example of stack error message[13]

Fig. 4. Two examples of posts (https://stackoverflow.com/questions/744735, https://stackoverflow.com/questions/59726805)

3. Data optimization. To improve the usability and accuracy of the dataset, we clean and optimize the dataset. The posts with large error message contain the fully qualified names of multiple APIs, but they do not belong to the API Post associating with 6 APIs as shown in Fig. 4(b). We find that API *java.lang.reflect.Method.invoke* and API *java.lang.Thread.run* will appear in such posts with high probability. Besides, posts related to no more than three APIs accounted for 83.05% of all posts. Based on the above observations, We select the larger proportion and more representative posts, and filter following types of posts to optimize the dataset: (1) the posts related to API *java.lang.reflect.Method.invoke* or API *java.lang.Thread.run*. (2) the posts related to more than 3 APIs. We unify the APIs in $< post, APIs >$ pairs into the format of *Package.Class.Method*. When using APIs' links for text matching, the purpose of retaining parameters is to ensure data accuracy as much as possible.

The JAPD: After data acquisition and optimization, we obtained the JAPD containing 35,825 $< post, APIs >$ pairs, and 34,160 pairs (95.35%) of the posts with code snippets. The JAPD is associated with 2704 Java APIs. There are 2008 APIs of which simple name are associated with more than one fully qualified name. In other word, 74% of the data is necessary for disambiguation. To make it easier for everyone to reuse the JAPD, we also provide the following data: APIs contained in all packages; the simple names associated with all fully qualified names.

4 Approach

4.1 Overview

To solve API recognition and disambiguation tasks, we propose a novel approach JARAD to infer the associated APIs in a post. Figure 5 is the architecture of the JARAD (**J**ava **A**PI **R**ecognition **A**nd **D**isambiguation), which mainly includes the following steps: (1) For each post, we perform data processing and parse out its text snippets and code snippets. The 'Title' of the post is treated as text. (2) Next, we train API mention recognition models for text and code respectively, obtain API mentions, and get candidate API set based on the prepared Java API set. If an API is a post-related API, it should be mentioned in the text snippets, the code snippets, or both. (3) Finally, we parse the post content again. The candidate API is scored by the frequency of the API type appearing in the post to infer API's fully qualified name. We perform API disambiguation based on the scores of candidate APIs. We will detail the two important steps of API recognition and API disambiguation in the JARAD.

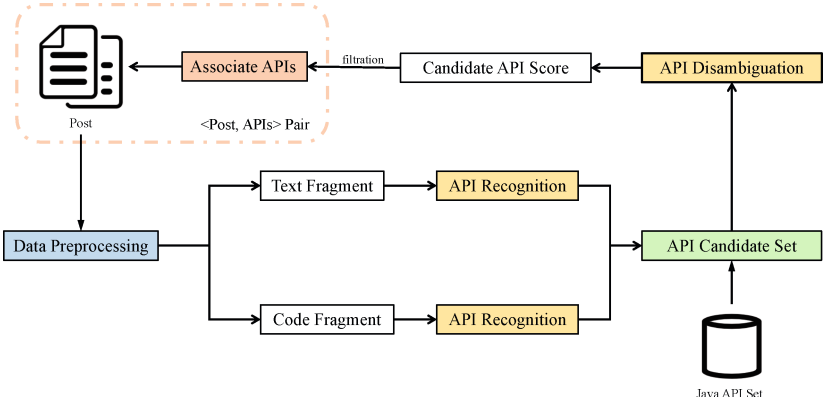


Fig. 5. The architecture of JARAD

4.2 API Mention Recognition

API mention recognition is implemented as the sequence labeling task. The model is shown in Fig. 6. We utilize Bi-directional Long Short-Term Memory (BiLSTM) and conditional random field (CRF) fusing context information to annotate API mentions in text and code. We first divide the text and code snippets into independent tokens, namely $\{T_1, T_2, T_3, ..., T_{n-1}, T_n\}$. Tokens in both text and code snippets are pre-labeled for training. 'B' and 'O' in the Fig. 6 represent API mentions and non-API mentions respectively. These tokens will be parsed forward and reverse by BiLSTM to solve the problem of long

sentence dependence. To more comprehensively integrate context information, we obtain the features of each Token, namely $\{F_1, F_2, F_3, ..., F_{n-1}, F_n\}$. As the output of BiLSTM layer, these features will be the input of the CRF layer. The CRF layer learns the constraint relationship of fragments and predicts the label of each Token through maximum likelihood estimation according to the idea that "the score of the real path is the highest among all the paths".

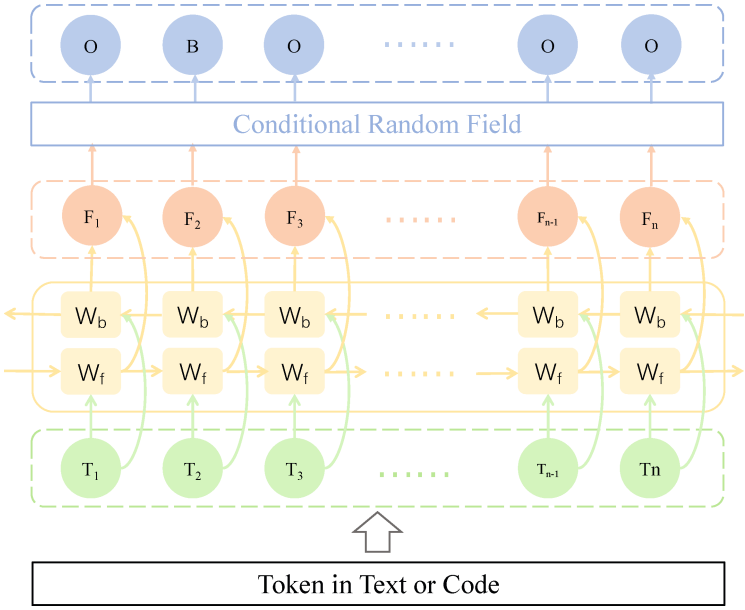


Fig. 6. API Mention Recognition Model Diagram

4.3 API Disambiguation

Aiming at the task of API disambiguation in informal texts in Stack overflow, Kien Luong et al. [13] proposed the DATYS+. It disambiguates Java API mentions via type scope when given API mentions. DATYS+ takes as input a Stack Overflow thread, API mentions and Java libraries, and outputs a set of APIs and their scores. DATYS+ first extracts the API candidate set from the Java library and API mentions. These candidate APIs are scored according to the frequency of the candidate API type (class or interface) appearing in different scopes of the Stack Overflow thread. DATYS+ considers four mention scopes: (1) Mention Scope, which covers the mention itself; (2) Text Scope, which covers text content, including mentions; (3) Code Scope, which covers code snippets. (4) Code Comments Scope, which covers code comments snippets. DATYS+ then rank the APIs based on the scores of the candidate APIs. Finally, the first

candidate API with a score more than 0 is selected as the associated API in this thread.

The API disambiguation method in this paper is improved on the basis of DATYS+. We leverage the frequency of the API type appearing in the above four mention scopes from API Post to score the API mention candidates set. However the DATYS+ filters irrelevant APIs through the 'Tags' in the post. That is, when the prefix of a API does not appear in the 'Tags', DATYS+ filters this candidate API. In our empirical study (see Sect. 2.3), we find that the 'Tags' of a post can only serve as an enhancement factor for linking APIs, but not a reason for filtering irrelevant APIs. In other words, the 'Tags' in a post can only enhance the probability that the candidate API is the correct post-related API. Based on this finding, we delete the step of using 'Tags' as the filter condition. We retain the idea of scoring the set of candidate APIs by the frequency of candidate APIs in the above four scopes. We finally take the APIs whose scores are not zero in the candidate set as the associated APIs of the posts.

5 Experiment Result

5.1 Experimental Settings

The experiments were conducted on an Ubuntu 18.04.5 LTS system equipped with 128GB memory and two Intel(R) Xeon(R) Silver 4210 CPUs. We train and evaluate the JARAD based on JAPD with 35,825 $< post, APIs >$ pairs. The JAPD is divided into a training set and a test set at a ratio of 5:1. We train separate models for text and code in posts on the API mention recognition task. It is noted that 79% of API Post are not with fully qualified names (see Sect. 2.3). The step of identifying API mentions (e.g. short names, aliases, etc.) allows our method to remain effectiveness in such posts.

5.2 Metrics

We use the unified evaluation metrics of sequence labeling tasks to evaluate the JARAD: Precision, Recall, and F1 measure. Precision represents the proportion of the correct APIs that the model predicts. Recall measures the number of APIs correctly inferred in the test set. F1 measure takes both Precision and Recall into account for evaluation. Their formulas are as follows:

$$Precision = \frac{True\ Positive}{True\ Positive + False\ Positive} \tag{1}$$

$$Recall = \frac{True\ Positive}{True\ Positive + False\ Negative} \tag{2}$$

$$F1 = \frac{2 * Precision * Recall}{Precision + Recall} \tag{3}$$

5.3 Experimental Results

We first evaluate the effects of the two main components of the JARAD. Due to the discrepancy between text and code on Stack Overflow, we train two API mention recognition models for text and code respectively. We also evaluate the effectiveness of the API disambiguation method which refers to the ability to link the correct API' fully qualified name given the correct API mentions. As shown in Table 3, our API mention recognition model achieves a Precision of 92.08% on code snippets, and our API disambiguation model achieves a Recall of 93.99%.

Table 3. Component evaluation results of the JARAD

Component	Precision	Recall	F1
API Recognition for Text	46.99%	69.77%	56.18%
API Recognition for Code	92.08%	84.24%	87.98%
API Disambiguation	78.88%	93.99%	85.78%

The overall performance is shown in Table 4. As suggested in the paper [2], our model favors Recall than Precision. So we try to tune model parameters with better Recall. Our JARAD has achieved a Precision of 71.58%, Recall of 76.84%, and F1 measure of 74.12%. In terms of Recall, our method outperforms ARCLIN [2] which achieves a Recall of 73.53%. Compared with ARCLIN as a SOTA model in the field of API mention recognition and disambiguation in informal texts, our method has the following two advantages: (1) ARCLIN works only on the text in the post while our method considers both text and code information; (2) ARCLIN aims at the APIs in the five libraries of Python while our method targets a wider range of dataset, expanding to all APIs in the Java JDK (see Sect. 5.4).

Table 4. Evaluation results of the JARAD

Approach	Precision	Recall	F1
APIReal [1]	78.7%	60.4%	68.3%
ARCLIN [2]	78.26%	73.53%	75.82%
JARAD	71.58%	**76.84%**	74.12%

5.4 Discussion

This section first compares the previous methods with the JARAD, highlights the advantages of JARAD, and then discusses the shortcomings of JARAD.

1. JARAD works on both text snippets and code snippets in posts. As we have observed empirically, 66.92% of API Post on Stack Overflow are mentioned in code snippets. Among the 35,825 pieces of data in our JAPD dataset, 34,160 pieces contain code snippets, accounting for 95% of the total. These phenomena emphasize to a certain extent that code snippets cannot be ignored in identifying API tasks on Stack Overflow. However, some studies [2,10] in recent years only considered text snippets. As a SOTA research based on the Python dataset, ARCLIN [2] discards the code snippets during data processing. Although the study [10] is also oriented to the Java dataset, it only considers the sentences in the post and ignores the code snippets.

2. JARAD aims at a wider range of dataset. Compared with some studies [1,2,5,10,12,13], our method targets the Java language and is applicable to a wider range of dataset containing all Java APIs, rather than targeting a few specific libraries. Due to the popularity of the two development languages Python and Java, researchers are more inclined to use the dataset about them. The data of ARCLIN [2] is based on five libraries of Python: Pytorch, Pandas, Tensorflow, Numpy, and Matplotlib. Their experiments also show that the accuracy of the generalization ability on Matplotlib is only 26.6% on average when using the other four libraries as the training set. Research [10] is based on Java APIs, but their dataset is only based on APIs' links. However, only 804 of our 110,171 unfiltered data are obtained through APIs' links. So we infer the certain extent that the amount of their research data is insufficient, and the number of APIs involved is also inadequate.

3. The structure of JARAD is simpler, and the training time of an epoch is shorter. The average time spent by the model inferring the API associated with a post is only 600 ms. Due to the different characteristics of text and code, we train API mention recognition models for them respectively (see Sect. 4.2). We find that it takes only 7 min for the text model to train one epoch and 28 min for the code model. This is the result of our training with CPU. Previous studies used the Bert fine-tuning model but did not elaborate on the training time of the model. To compare the training time of the Bert fine-tuning model with our model, we add the Bert component before the BiLSTM layer of the API mention recognition model in JARAD. We find that it takes 18 min for the text model to train one epoch, and takes one and a half hours for the code model, which is significantly longer than the training time of our JARAD model.

Threats to Validity: The JARAD proposed in this paper experiments on the JAPD. We extract the Java API Post to construct the JAPD. The API Post is associated with the API in the official Java JDK documentation. But the API recognition model we proposed cannot distinguish the source of the API. As shown in Fig. 7, when we use the API recognition model trained based on the text snippets in the post to infer the API mentions associated with the text snippets, the model incorrectly recognizes 'drawloop()' as an API mention. But in fact, this is a user-defined method (shown in the code snippet in the post). We infer that both the name 'drawloop' and the brackets of the method

are the reasons for the false positives in our API mention recognition model. And 'e.drawloop()' is not recognized as an API mention because our model recognizes that the prefix 'e' of the method is not a valid package or class name. It is undeniable that our model has room for improvement.

Java Graphics Error: Static / Nonstatic mishap

Asked 8 years, 8 months ago Modified 8 years, 7 months ago Viewed 358 times

2

So, I was working on a project of slots, because I wanted to get familiar with the java graphics library. The setup worked rather flawlessly. Then, I tried drawing a line to test the graphics, and got the error:

```
engine.java:9: non-static method drawLine(int,int,int,int) cannot be referenced
from a static context
        java.awt.Graphics.drawLine(1, 2, 11, 12);
                          ^
1 error
```

I went with the advice of my friend and created a new `engine` and named it `e`, then instead of doing `drawloop()` I did `e.drawloop`, but got the same error.

Code:

```
import java.awt.Graphics;
import java.awt.Dimension;
import javax.swing.JComponent;
import javax.swing.JFrame;
import javax.swing.SwingUtilities;

public class engine{
    public void drawloop(){
        java.awt.Graphics.drawLine(1, 2, 11, 12);
    }
    public static void main(String[] args){
          SwingUtilities.invokeLater(new Runnable()
                {
                 public void run()
                 {
                    display();
                 }
                });
        engine e = new engine();
        e.drawloop();
    }
```

Fig. 7. An example of a customed (https://stackoverflow.com/questions/22889139)

Another fact is that the improvement of the JARAD model is not significant enough. Compared with ARCLIN [2], our model is not outstanding in terms of numerical results, which may be the reason for the simpler model structure. But our model has a clear and concise structure, which is easy to understand and explain. We reduce the training cost and get the final model quickly.

6 Related Work

6.1 API Recognition and Disambiguation Dataset

Datasets are the basis for researchers to train and test models. However, the lack of available and sufficient dataset forces researchers in the field of API

recognition and disambiguation in informal texts to use manual labeling methods to construct [2,5,12,13]. Although ARCLIN [2] eliminates the manual labeling step when training the model, it still have to manually label the fully qualified name of the API during testing. Study [5] manually flagged API mentions(600 per lib) in 3600 Stack Overflow posts for six libraries (three Python libraries: Pandas, NumPy and Matplotlib; one Java library: JDBC, one JavaScript library: React, and one C library: OpenGL) for experiments. ARSeek [13] used 380 Stack Overflow threads as their dataset from the study [12]. DATYS [12] regards the questions and corresponding answers on Stack Overflow as threads and manually marks 380 pieces of data, which are associated with the APIs of four Java libraries (Guava, Mockito, AssertJ, and Fastjson). It is worth noting that the proposers of the DATYS method also highlighted that 'there is no available ground truth dataset of Java API method mentions in Stack Overflow' is the reason for their manual labeling of the dataset.

6.2 API Mention Recognition and Disambiguation Research

The research methods of API mention recognition and disambiguation in informal texts are mainly based on rules [3,4,17] and machine learning [1,2,9,10,12,13]. Bacchelli et al. [17] proposed a rule-based approach to extract API mentions from emails by designing regular expressions applicable to different languages. They then utilized two string-match information retrieval techniques (i.e., vector space model and LSI) for API disambiguation. Treude et al. [3] identify API mentions by using different regular expressions for questions and text in Stack Overflow posts. There is also a method [4] that utilizes compound camel-case terms to identify code elements in the text, but this method does not consider the confusion of common words and APIs. APIReal [1] proposes a semi-supervised machine learning approach to recognize API mentions and utilizes heuristics methods (including mention-mention similarity, scope filtering, and mention-entry similarity) for API disambiguation. ARCLIN [2] performs recognition and disambiguation based on neural networks and similarity comparisons. Different from ARCLIN, the method [10] utilizes deep learning methods for both recognition and disambiguation. Because the API mention recognition and disambiguation research in informal texts can be subdivided into two steps, some research will study a certain step independently. For example, the methods of DATYS [12] and ARSeek [13] assume known API mentions and determine the fully qualified names of the mentioned APIs by studying the text and code snippets in the discussion, which is API disambiguation research [1,17,23–28]. API mention recognition and disambiguation Research can support downstream tasks such as API recommendation [15,16,33], API Mining [11,15,16,18,19], API misuse detection [34,35], and API resource retrieval [29–32,36].

7 Conclusion and Future Work

Stack Overflow is one of the most popular communities for spontaneous discussions among developers. Browsing past related posts on the community and

launching a new post to ask for help are common means for them to solve problems on API invoking. Reasoning about the APIs discussed in previous posts can help them search and filter the massive existing posts to quickly find related APIs or similar problems. This inference task can also help downstream tasks such as API Recommendation and API Mining. This paper conducts an empirical survey of posts on Stack Overflow to study the characteristics of APIs in discussions. Then, according to the APIs' fully qualified names and links in the API official documentation, the dataset of more than 30,000 $< post, APIs >$ pairs is automatically constructed based on the method of text matching. Finally, the approach JARAD is proposed to solve the tasks of API mention recognition and disambiguation linking. Compared with previous methods, the structure of the JARAD used in this paper is simpler and JARAD has a good performance on Recall. In the future, we can extend the research in several aspects. We will improve our method of obtaining the dataset, expand the types of development language, and obtain more general API recognition and disambiguation datasets in informal texts. We will also improve API recognition and disambiguation methods to adapt to more general data.

Acknowledgements. This work was supported in part by the High Performance Computing Center of Central South University.

References

1. Ye, D., Bao, L., Xing, Z., et al.: APIReal: an API recognition and linking approach for online developer forums. Empir. Softw. Eng. **23**, 3129–3160 (2018)
2. Huo, Y., Su, Y., Zhang, H., et al.: ARCLIN: automated API mention resolution for unformatted texts. In: Proceedings of the 44th International Conference on Software Engineering, pp. 138–149 (2022)
3. Treude, C., Robillard, M.P.: Augmenting API documentation with insights from stack overflow. In: Proceedings of the 38th International Conference on Software Engineering, pp. 392–403 (2016)
4. Rigby, P.C., Robillard, M.P.: Discovering essential code elements in informal documentation. In: 2013 35th International Conference on Software Engineering (ICSE), pp. 832–841. IEEE (2013)
5. Ma, S., Xing, Z., Chen, C., et al.: Easy-to-deploy API extraction by multi-level feature embedding and transfer learning. IEEE Trans. Software Eng. **47**(10), 2296–2311 (2019)
6. Ye, D., Xing, Z., Foo, C.Y., et al.: Learning to extract API mentions from informal natural language discussions. In: 2016 IEEE International Conference on Software Maintenance and Evolution (ICSME), pp. 389–399. IEEE (2016)
7. Ge, C., Liu, X., Chen, L., et al.: Make it easy: an effective end-to-end entity alignment framework. In: Proceedings of the 44th International ACM SIGIR Conference on Research and Development in Information Retrieval, pp. 777–786 (2021)
8. Ye, D., Xing, Z., Foo, C.Y., et al.: 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, pp. 90–101. IEEE (2016)

9. Chen, C., Xing, Z., Wang, X.: Unsupervised software-specific morphological forms inference from informal discussions. In: 2017 IEEE/ACM 39th International Conference on Software Engineering (ICSE), pp. 450–461. IEEE (2017)
10. Yin, H., Zheng, Y., Sun, Y., et al.: An API learning service for inexperienced developers based on API knowledge graph. In: 2021 IEEE International Conference on Web Services (ICWS), pp. 251–261. IEEE (2021)
11. Baltes, S., Treude, C., Diehl, S.: SOTorrent: studying the origin, evolution, and usage of stack overflow code snippets. In: 2019 IEEE/ACM 16th International Conference on Mining Software Repositories (MSR), pp. 191–194. IEEE (2019)
12. Luong, K., Thung, F., Lo, D.: Disambiguating mentions of API methods in stack overflow via type scoping. In: 2021 IEEE International Conference on Software Maintenance and Evolution (ICSME), pp. 679–683. IEEE (2021)
13. Luong, K., Hadi, M., Thung, F., et al.: ARSeek: identifying API resource using code and discussion on stack overflow. In: Proceedings of the 30th IEEE/ACM International Conference on Program Comprehension, pp. 331–342 (2022)
14. Luong, K., Thung, F., Lo, D.: ARSearch: searching for API related resources from stack overflow and GitHub. In: Proceedings of the ACM/IEEE 44th International Conference on Software Engineering: Companion Proceedings, pp. 11–15 (2022)
15. Huang, Q., Xia, X., Xing, Z., et al.: API method recommendation without worrying about the task-API knowledge gap. In: Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering, pp. 293–304 (2018)
16. Rahman, M.M., Roy, C.K., Lo, D.: RACK: automatic API recommendation using crowdsourced knowledge. In: 2016 IEEE 23rd International Conference on Software Analysis, Evolution, and Reengineering (SANER), vol. 1, pp. 349–359. IEEE (2016)
17. Bacchelli, A., Lanza, M., Robbes, R.: Linking e-mails and source code artifacts. In: Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering, vol. 1, pp. 375–384 (2010)
18. Liu, M., Peng, X., Marcus, A., et al.: API-related developer information needs in stack overflow. IEEE Trans. Software Eng. **48**(11), 4485–4500 (2021)
19. Velázquez-Rodríguez, C., Constantinou, E., De Roover, C.: Uncovering library features from API usage on Stack Overflow. In: 2022 IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER), pp. 207–217. IEEE (2022)
20. Singh, R., Mangat, N.S.: Elements of Survey Sampling. Springer, Dordrecht (2013). https://doi.org/10.1007/978-94-017-1404-4
21. Li, H., Li, S., Sun, J., et al.: Improving API caveats accessibility by mining API caveats knowledge graph. In: 2018 IEEE International Conference on Software Maintenance and Evolution (ICSME), pp. 183–193. IEEE (2018)
22. Wang, C., Peng, X., Liu, M., et al.: A learning-based approach for automatic construction of domain glossary from source code and documentation. In: Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering, pp. 97–108 (2019)
23. Antoniol, G., Canfora, G., Casazza, G., et al.: Recovering traceability links between code and documentation. IEEE Trans. Software Eng. **28**(10), 970–983 (2002)
24. Dagenais, B., Robillard, M.P.: Recovering traceability links between an API and its learning resources. In: 2012 34th International Conference on Software Engineering (ICSE), pp. 47–57. IEEE (2012)
25. Marcus, A., Maletic, J.I.: Recovering documentation-to-source-code traceability links using latent semantic indexing. In: 25th International Conference on Software Engineering, 2003. Proceedings, pp. 125–135. IEEE (2003)

26. Phan, H., Nguyen, H.A., Tran, N.M., et al.: Statistical learning of API fully qualified names in code snippets of online forums. In: Proceedings of the 40th International Conference on Software Engineering, pp. 632–642 (2018)
27. Saifullah, C.M.K., Asaduzzaman, M., Roy, C.K.: Learning from examples to find fully qualified names of API elements in code snippets. In: 2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE), pp. 243–254. IEEE (2019)
28. Subramanian, S., Inozemtseva, L., Holmes, R.: Live API documentation. In: Proceedings of the 36th International Conference on Software Engineering, pp. 643–652 (2014)
29. Nguyen, T., Tran, N., Phan, H., et al.: Complementing global and local contexts in representing API descriptions to improve API retrieval tasks. In: Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering, pp. 551–562 (2018)
30. Ye, X., Shen, H., Ma, X., et al.: From word embeddings to document similarities for improved information retrieval in software engineering. In: Proceedings of the 38th International Conference on Software Engineering, pp. 404–415 (2016)
31. Rój, M.: Exploiting user knowledge during retrieval of semantically annotated API operations. In: Proceedings of the Fourth Workshop on Exploiting Semantic Annotations in Information Retrieval, pp. 21–22 (2011)
32. Zhou, Y., Wang, C., Yan, X., et al.: Automatic detection and repair recommendation of directive defects in Java API documentation. IEEE Trans. Software Eng. **46**(9), 1004–1023 (2018)
33. Xie, W., Peng, X., Liu, M., et al.: 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, pp. 1015–1026 (2020)
34. Ren, X., Sun, J., Xing, Z., et al.: 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, pp. 925–936 (2020)
35. Ren, X., Ye, X., Xing, Z., et al.: API-misuse detection driven by fine-grained API-constraint knowledge graph. In: Proceedings of the 35th IEEE/ACM International Conference on Automated Software Engineering, pp. 461–472 (2020)
36. Li, J., Sun, A., Xing, Z., et al.: API caveat explorer–surfacing negative usages from practice: an API-oriented interactive exploratory search system for programmers. In: The 41st International ACM SIGIR Conference on Research & Development in Information Retrieval, pp. 1293–1296 (2018)