Empir Software Eng
https://doi.org/10.1007/s10664-018-9608-7

APIReal: an API recognition and linking approach for online developer forums

Deheng Ye[1,2] · Lingfeng Bao[3] · Zhenchang Xing[4] · Shang-Wei Lin[1]

Abstract When discussing programming issues on social platforms (e.g, Stack Overflow, Twitter), developers often mention APIs in natural language texts. Extracting API mentions from natural language texts serves as the prerequisite to effective indexing and searching for API-related information in software engineering social content. The task of extracting API mentions from natural language texts involves two steps: 1) distinguishing API mentions from other English words (i.e., *API recognition*), 2) disambiguating a recognized API mention to its unique fully qualified name (i.e., *API linking*). Software engineering social content lacks consistent API mentions and sentence writing format. As a result, API recognition and linking have to deal with the inherent ambiguity of API mentions in informal text, for example, due to the ambiguity between the API sense of a common word and the normal sense of the word (e.g., append, apply and merge), the simple name of an API can map to several APIs of the same library or of different libraries, or different writing forms

Communicated by: Denys Poshyvanyk

✉ Lingfeng Bao
lingfengbao@zju.edu.cn

Deheng Ye
ydyl1991@gmail.com

Zhenchang Xing
zhenchang.xing@anu.edu.au

Shang-Wei Lin
shang-wei.lin@ntu.edu.sg

1 School of Computer Science and Engineering, Nanyang Technological University, 50 Nanyang Avenue, Singapore, Singapore

2 Tencent AI Lab, Shenzhen, China

3 College of Computer Science, Zhejiang University, Hangzhou, China

4 Research School of Computer Science, Australian National University, Canberra, Australia

Published online: 05 March 2018

of an API should be linked to the same API. In this paper, we propose a semi-supervised machine learning approach that exploits name synonyms and rich semantic context of API mentions for API recognition in informal text. Based on the results of our API recognition approach, we further propose an API linking approach leveraging a set of domain-specific heuristics, including mention-mention similarity, scope filtering, and mention-entry similarity, to determine which API in the knowledge base a recognized API actually refers to. To evaluate our API recognition approach, we use 1205 API mentions of three libraries (*Pandas, Numpy, and Matplotlib*) from Stack Overflow text. We also evaluate our API linking approach with 120 recognized API mentions of these three libraries.

Keywords Mining software repositories · API recognition · API linking · Semi-supervised learning · Natural language processing

1 Introduction

APIs are an important resources for software engineering. APIs appear not only in source code, but also in natural language texts, such as formal API specifications and tutorials, as well as developers' informal discussions, e.g., emails and online Q&A posts. Extracting API mentions from natural language texts serves as the prerequisite to effective indexing and searching for API-related information in software engineering social content. With the advances of Web 2.0, Stack Overflow has become the most prominent online forum in software engineering, archiving rich informal textual discussions on APIs. In this paper, we are concerned with the problem of extracting API mentions from informal texts such as Stack Overflow discussions.

Extracting API mentions from natural language texts involves two consecutive steps: *API recognition* and *API linking*. API recognition distinguishes API mentions from normal English words in a sentence. We illustrate the API recognition step in Fig. 1, which shows a sentence from a Stack Overflow post (postid is 12182744). In this sentence, we would like to recognize the *series* and the second *apply* as API mentions, which are a class name and a method name of the Pandas Library, respectively. In contrast, the first *apply* should be recognized as a common English word. After an API mention has been recognized, the API linking step links it to its unique fully qualified form as appeared in the official API documentation. The API linking step is illustrated in Fig. 2. In this example, after we recognize the API mention "apply" in the sentence shown in Fig. 1, API linking aims to further determine which *apply* method this API mention actually refers to, because there are more than one method with the simple name "apply" in the Pandas library.

Informal discussions on social platforms (such as Stack Overflow) are contributed by millions of users with very diverse technical and linguistic background. Such informal texts

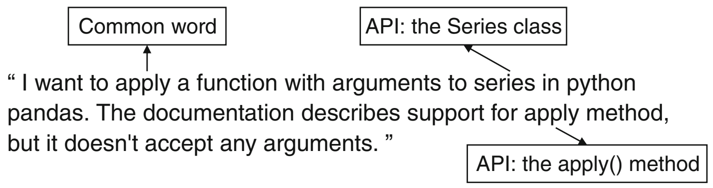

Common word
API: the Series class
" I want to apply a function with arguments to series in python pandas. The documentation describes support for apply method, but it doesn't accept any arguments. "
API: the apply() method

Fig. 1 Illustrating our task. Step 1: API recognition

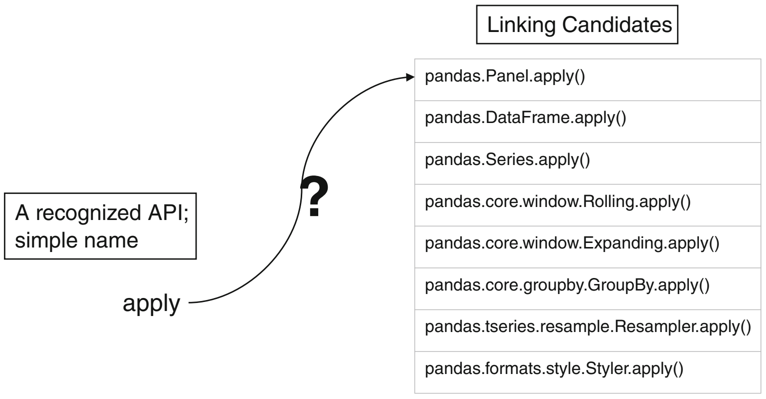

Linking Candidates
A recognized API; simple name
apply
?
pandas.Panel.apply()
pandas.DataFrame.apply()
pandas.Series.apply()
pandas.core.window.Rolling.apply()
pandas.core.window.Expanding.apply()
pandas.core.groupby.GroupBy.apply()
pandas.tseries.resample.Resampler.apply()
pandas.formats.style.Styler.apply()

Fig. 2 Illustrating Our Task. Step 2: API Linking

lack consistent API mention and sentence writing format. As shown in Table 1 and Fig. 1, fully qualified API names are rarely mentioned in developers' discussions on Stack Overflow. Instead, Stack Overflow users often use the simple name or other derivational forms of an API in the discussions. Furthermore, as shown in Table 2, even for the same API form, the surrounding sentence context of an API mention (e.g., the usage of verb, noun and preposition) also varies greatly. As a result, API recognition and linking have to deal with the inherent ambiguities of API mentions in informal text. For example, we need to distinguish the API sense of the word "apply" and the normal sense of "apply" in the sentence shown in Fig. 1. We need to disambiguate the specific *apply* method that the simple-name API mention "apply" in a sentence actually refers to as illustrated in Fig. 2. The challenge is that these ambiguities have to be resolved in the face of API mention and sentence context variations as shown in Tables 1 and 2.

In this paper, we develop a semi-supervised machine learning approach to solve the API recognition problem in informal texts. To model API-mention and sentence-context variations, our approach exploits state-of-the-art unsupervised language models, in particular class-based Brown Clustering (Brown et al. 1992; Liang 2005) and neural-network-based word embedding (Mikolov et al. 2013b; Turian et al. 2010) to learn word clusters of semantically similar words from the abundant unlabeled text. Empowered by the compound word-cluster features from unsupervised language models fed into a linear-chain Condition

Table 1 A subset of variant forms of API mentions

Writing form	Frequency	Remarks
pandas.DataFrame.apply()	3	Standard form
pandas.DataFrame.apply	10	
.apply	624	Nonpolysemous derivations
apply()	177	
.apply()	79	
dataframe.apply	117	
df.apply	215	
df.apply()	20	
apply	4,530	Polysemy

Table 2 Sentences mentioning the same API form

Post ID	Sentence-context variations
15589354	I have finally decided to use `apply` which I understand is more flexible.
29627130	if you run `apply` on a series the series is passed as a np.array.
25275009	It is being run on each row of a Pandas DataFrame via the `apply` function.
21390035	I am confused about this behavior of `apply` method of groupby in pandas.
18524166	You are looking for `apply`.
7580456	I tested with `apply`, it seems that when there are many sub groups, it's very slow.

Random Field (CRF) model (Lafferty et al. 2001), together with an iterative self-training mechanism (a.k.a. bootstrapping) (Wu et al. 2009), our approach requires a small set of human labeled sentences to train a robust model for recognizing API mentions in informal natural language sentences.

Based on the results of our API recognition approach, we develop a rule-based fine-grained API linking method. Our API linking approach leverages mention-mention similarity, mention-entry similarity, and a set of Stack Overflow specific heuristics for scope filtering. We borrow the terms *mention-mention similarity* and *mention-entry similarity* from the entity linking work done by Liu et al. (2013). Specifically, mention-mention similarity measures the contextual similarity between recognized API mentions, and mention-entry similarity computes the similarity between the context of the current API mention and the content of an API entry in the knowledge base. Scope filtering uses a set of Stack Overflow specific information, such as tags, question title and code blocks, to help eliminate the number of linking candidates.

For the evaluation of our API recognition approach, we deploy our approach on Stack Overflow discussions that contain APIs of three Python libraries, i.e., Pandas, Numpy and Matplotlib. We choose these Python libraries, because they define many common-word APIs, making their informal mentions ambiguous with the normal sense of the common words. Meanwhile, these three libraries are popular Python libraries for very different functionalities and have been widely discussed on Stack Overflow. We then compare our approach with three state-of-the-art API recognition methods in natural language texts, including lightweight regular expressions (Bacchelli et al. 2010), code-annotation enhanced regular expressions (Parnin et al. 2012; Linares-Vásquez et al. 2014), and machine-learning based software-specific entity recognition (Ye et al. 2016a). We show that our approach consistently and significantly outperforms these three baseline methods. For the evaluation of our API linking approach, we first prepare a knowledge base containing the information of the official fully qualified API names of the Python Standard Library and the aforementioned three Python libraries. Then, based on the results of our API recognition, we examine the API linking results of 120 ambiguous API mentions that have more than 1 linking candidates in the knowledge base. We show that our linking approach achieves high precision and recall.

2 Challenges in API recognition and linking in informal text

In this section, we describe the challenges for effective API recognition and linking in informal software engineering texts.

API recognition This step aims to recognize tokens in a natural language sentence that refer to public modules, classes, methods or functions of certain libraries as API mentions. API mentions exist in both source code and natural language texts. In this work, we perform API recognition and linking from natural language texts. Extracting APIs from code snippets using code parsers, e.g., partial program analyzer (PPA), is related to but not the research focus of the present paper.

Recognizing API mentions in natural language sentences is a prerequisite for indexing, analyzing, and searching informal natural language discussions for software engineering tasks (Rigby and Robillard 2013). Many applications require API recognition as the first step or can benefit from it, e.g., fine-grained API linking (Bacchelli et al. 2010; Dagenais and Robillard 2012; Rigby and Robillard 2013; Subramanian et al. 2014), API recommendation (Rahman et al. 2016; Zheng et al. 2011) and bug fixing (Chen and Kim 2015; Gao et al. 2015). Indeed, the importance of API recognition from natural language sentences has long been recognized. Representative techniques include language convention based regular expressions (Bacchelli et al. 2009, 2010; Dagenais and Robillard 2012) and island parsing (Bacchelli et al. 2011; Rigby and Robillard 2013). These techniques usually rely on observational orthographic heuristics to distinguish APIs from normal words in a natural language sentence: 1) distinct API naming conventions (e.g., words containing camelcases or special characters like ., ::, or ()); 2) structured sentence format (e.g., code-like phrases like “a=series.apply()” or API annotation). These heuristics perform well for cases where orthographic features are prevalent and consistently used, e.g., extracting *camelcased* Java APIs from *official* documentations (Dagenais and Robillard 2012). However, they fall short to address the following two fundamental challenges in API recognition from *informal* natural language texts:

– *Common-word polysemy:* Many APIs’ simple name is a single common word. For example, 55.04% of the *Pandas*’s APIs have common-word simple names, such as the class *Series* and the method *apply*. When such APIs are mentioned with distinct orthographic features, such as *pandas.Series*, *<code>apply</code>*, and *apply()*, we can easily recognize them. Unfortunately, this is not always the case due to the informal nature of the texts. Then, such common-word APIs appear just as common words in a sentence, as shown in Fig. 1. In fact, the token *apply* (using a software-specific tokenizer (Ye et al. 2016a)) appears 4,530 times in the discussions tagged with `pandas`. Using our trained model, we estimate that about 35.1% (1590/4530) of these *apply* tokens are true API mentions (labeling confidence score $>$ 0.8). This creates the challenge in *distinguishing the API sense of a common word from the normal sense of the word*, for example, the first *apply* (a common word) and the second *apply* (an API mention) in Fig. 1. In fact, for API recognition from developers’ informal discussions (e.g., emails). Bacchelli et al. (2010) have already shown that this challenge poses a big threat to language-convention based regular expressions, as no observational orthographic features can be utilized. The polysemy problem becomes even more evident for the case of the C programming language due to its API naming convention, when compared to Java, as studied in Bacchelli et al. (2010). However, this common-word polysemy challenge is generally avoided by considering only APIs mentions with distinct orthographic features in an existing work (Rigby and Robillard 2013). As to our best knowledge, there has been no research work in the software engineering community addressing the polysemy problem in fine-grained API recognition.
– *Sentence-format variations:* Informal discussions on social platforms (such as Stack Overflow) are contributed by millions of users with very diverse technical and linguistic

background. Such informal discussions are full of misspellings, synonyms, inconsistent annotations, etc. Consequently, the same API is often mentioned in many different forms intentionally or accidentally. Table 1 lists a subset of variant forms of potential mentions of the *apply* method and their frequencies in the discussions tagged with `pandas`. We can see that standard API names are mentioned very few times. Instead, users use non-standard synonyms (e.g., *DataFrame* written as *df*) and various non-polysemous derivational forms (e.g., *.apply*, *df.apply*) that can be partially matched to the full name or the full name synonym. In addition, the polysemous common-word *apply* is used 4,530 times. Even for the same API form, the surrounding sentence context of an API mention also varies greatly. As shown in Table 2, there are lacks of consistent usage of verb, noun and preposition in the discussions. All these API-mention and sentence-context variations make it extremely challenging to *develop a complete set of regular expressions or island grammars for inferring API mentions*.

To handle common-word polysemy and API-mention variations, we propose to exploit the sentence context in which an API is mentioned to recognize API mentions in informal natural language sentences. The rationale is that no matter what an API's name is or in what form an API is mentioned, the sentence context of an API mention can help distinguish an API mention from non-API words. However, as shown in Table 2, to make effective use of sentence context, we must model sentence-context variations in informal social discussions. Unfortunately, it is impractical to develop a complete set of sentence context rules or to label a huge amount of data to train a machine learning model, not only due to prohibitive effort needed but also out-of-vocabulary issue (Li and Sun 2014; Liu et al. 2011) in informal text, i.e., variations that have not been seen in the training data even after a huge amount of data has been examined.

API linking After an API mention has been recognized, API linking step further links the API mention to its unique fully qualified form as appeared in the official API documentation. As pointed out in Table 1, Stack Overflow users rarely write the fully qualified name of an API. Instead, the unqualified informal name has been widely used, particularly the simple name. The partially qualified name and simple name of an API can be ambiguous (Dagenais and Robillard 2012; Subramanian et al. 2014). In 2014, Subramanian et al. observed 89% of the Java unqualified method names collide. The ambiguity in API mentions leads to the fact that even after we recognize a word as a true API mention, we may still not know which specific API the API mention refers to.

An API linking system should be able to handle the inherent ambiguities of API mentions. In 2012, Dagenais and Robillard summarized four kinds of ambiguities lying in the supporting documents of the Spring framework written in Java (including both natural language texts and code snippets). In 2014, Subramanian et al. highlighted two kinds of ambiguities in the Java code snippets from Stack Overflow. In our context, we are concerned with linking API mentions in informal natural language texts from Stack Overflow posts (excluding code snippets), and we use Python APIs as a case study. The ambiguities of Python API mentions in Stack Overflow text are similar but also different to those existing works, discussed as follows.

– *Declaration Ambiguity.* We reuse the term "declaration ambiguity" from Dagenais and Robillard (2012) and Subramanian et al. (2014). Fully qualified API names are rarely mentioned in developers' discussions on Stack Overflow. Instead, Stack Overflow users often use the simple name or other derivational forms of an API in a post, as shown in Table 1 and Fig. 1. In such cases, it is not an easy task for the machine to automatically and precisely infer the correct linking target, i.e., a specific API that an ambiguous API mention refers to.

- *External Reference Ambiguity.* We continue to use the term "external reference ambiguity" same as Dagenais and Robillard (2012) and Subramanian et al. (2014). One Stack Overflow discussion thread can cover knowledge of more than one library. As such, Stack Overflow text may refer to an API declared in an external library but has the same simple name or partially qualified name as one of the APIs in the focused libraries. For example, *Pandas* is one of our studied libraries, Stack Overflow discussions on *Pandas* may refer to an API in the Python Standard Library, or other related libraries.
- *Synonym Ambiguity.* As mentioned above, the informal nature of online Q&A discussions introduces synonyms, abbreviations, misspellings (errors), inconsistent annotations, etc. Consequently, the same API is often mentioned in various forms, while these variant writing forms should be linked to the same API in the knowledge base. The ambiguity of such variant API mentions is different from the *language ambiguity* defined in Dagenais and Robillard (2012) which only covers errors. Thus, we name it synonym ambiguity.

3 Related work

3.1 API recognition and linking in software engineering

Our work is related to both API recognition and API linking. API recognition is a foundational software engineering task. It is the prerequisite to the research on API linking (Antoniol et al. 2002; Marcus et al. 2003; Jiang et al. 2008; Bacchelli et al. 2009, 2010; Dagenais and Robillard 2012; Rigby and Robillard 2013; Subramanian et al. 2014). It can also benefit many other software-related applications, such as API recommendation (Rahman et al. 2016; Zheng et al. 2011), bug fixing (Chen and Kim 2015; Gao et al. 2015), and API usage tutorial linking (Wu et al. 2016). Here we discuss some representative methods for recognizing and linking APIs, which are categorized and summarized in Table 3.

Research on API recognition and linking can be roughly divided into two categories based on the linking granularity, i.e., coarse-grained and fine-grained. The trend is evolving from coarse-grained to more fine-grained. Coarse-grained link refers to the link between two coarse-grained artifacts like an entire document and a source class in the source code. One important reason of creating coarse-grained links is to ease software maintenance by bridging code blocks, e.g., class, to the corresponding documentation. Therefore, we can consider these coarse-grained linking works as software maintenance oriented. By comparison, fine-grained recognition and linking attempt to recognize a single API that consists of one token and link it to its formal form. Developers can acquire more specific knowledge of an API from applications based on such fine-grained linking techniques, e.g., the Baker tool (Subramanian et al. 2014). In this sense, these fine-grained linking works could be considered as knowledge acquisition oriented.

Antoniol et al. (2002), Marcus et al. (2003), and Jiang et al. (2008) performed coarse-grained API linking leveraging information retrieval techniques, i.e., vector space model (VSM), latent semantic indexing (LSI) and incremental LSI, respectively, to link the textual content of API documentation to source code of targeted systems. In these work, there are no advanced fine-grained API recognition techniques involved, essentially the API mentions are preprocessed together with other texts and are represented as bag-of-words to be utilized by information retrieval methods.

Table 3 Comparing with existing api recognition and linking methods

	Prior Work	Artifacts Studied	Key Techniques		Granularity
			API recognition	API linking	
Software maintenance -oriented	(Antoniol et al. 2002)	API documentation	Text preprocessing	Vector Space Model	Coarse-grained
	(Marcus et al. 2003)	API documentation	Text preprocessing	Latent Semantic Indexing	Coarse-grained
	(Jiang et al. 2008)	API documentation	Text preprocessing	Incremental LSI	Coarse-grained
	(Bacchelli et al. 2009, 2010)	Developer emails	Regular expressions	String matching, IR tech. (VSM, LSI)	Coarse-grained
	(Bacchelli et al. 2011)	Developer emails	Island grammar	–	Coarse-grained
Knowledge acquisition -oriented	(Dagenais and Robillard 2012)	Tutorial, forum text, code snippet	Regular expressions, PPA	Filtering heuristics based on recognition results	Fine-grained
	(Rigby and Robillard 2013)	Online text	Island grammar	Filtering heuristics based on recognition results	Fine-grained
	(Subramanian et al. 2014)	Online code snippet	PPA	Iteratively filter Oracle	Fine-grained
	APIReal	Online text	Semi-supervised learning	Textual similarity, scope filter	Fine-grained

Bacchelli et al. (2009, 2010) developed an API extraction and linking infrastructure (referred to as Miler). They used lightweight regular expressions of distinct orthographic features and information retrieval techniques to detect class and method mentions in developer emails. They found that information retrieval techniques did not work for fine-grained API extraction task, whose performance was even significantly worse than lightweight regular expressions. Importantly, their study pointed out that common-word polysemy and non-standard API synonyms significantly degraded the performance of lightweight regular expressions. However, no working solution tackling the polysemy and synonym problems in API recognition is proposed.

Dagenais and Robillard (2012) developed RecoDoc to extract Java APIs from several learning resources (formal API documentation, tutorial, forum posts, code snippets) and then performed traceability link recovery over the contents of these different sources. They devised a pipeline of filters to resolve the traceability link ambiguities. However, they extracted API mentions from natural language texts using regular expressions similar to those of Miler (Bacchelli et al. 2010). That is, their API extraction from natural language texts again relied on distinct orthographic features of APIs.

Island parsing is another popular technique for extracting information of interest from texts. By defining island grammars, the textual content is separated into constructs of interest (island) and the remainder (water) (Moonen 2001). Bacchelli et al. (2011) extracted structured code fragments from natural language texts with island parsing. However, the recognized code fragments were not further parsed to fine-grained APIs and no API linking is conducted. Rigby and Robillard (2013) also used island parser to identify code-like elements that can potentially be APIs. They further resolved the code-like phrases to fine-grained APIs. However, they assumed that true API mentions had to be written with orthographic features to be captured by island grammars. In their study, simple names of methods that are not suffixed by () are simply ignored (Rigby and Robillard 2013). For example, they considered only *HttpClient.execute* and *execute()* as API mentions, but ignored the single word *execute* which also likely refers to the same API.

There is much auxiliary information that can help users identify code or APIs in the web page of a Stack Overflow post (e.g., the code snippets inside $< code >< /code >$, API links inside $< a >< /a >$, etc.). For example, both of the two studies by Parnin et al. (2012) and Linares-Vásquez et al. (2014) utilize HTML tags including $< code >< /code >$ and $< a >< /a >$ to extract and link the API entities on Stack Overflow. Using the extracted API mentions, Parnin et al. (2012) studied the phenomena of crowd documentation and Linares-Vásquez et al. (2014) studied the impact of API changes on Stack Overflow discussions. Subramanian et al. (2014) developed the Baker tool to extract API mentions from online code snippet on Stack Overflow using partial program analysis (PPA), and then linked the extracted APIs to their formal form that appeared on the offical documentation. The core idea of Baker is to utilize the co-occurance of APIs in the same code snippet to iteratively reduce the number of linking candiates in the knowledge base. The artifact studied in Baker is code snippet, while we are concerned with API recognition and linking in informal natural language texts.

3.2 Entity recognition and linking in general domains

Our approach for API recognition is related to two lines of research in the natural language processing community, i.e., named entity recognition (NER) whose goal is to recognize and categorize entities (e.g., person, location and organization) in natural language texts (Liu et al. 2011; Li and Sun 2014; Liao and Veeramachaneni 2009), and word sense disambiguation (WSD) whose goal is to disambiguate the sense of polysemous words in a given sentence context (Mihalcea 2004; Navigli 2009; Chen et al. 2014). In our dataset, we observe the sense ambiguity of normal English words (for example "*set* as a verb or a noun") and the presence of co-reference (like "it computes the length of a set in which it refers to an API"). In this work, our focus is to disambiguate the API sense of a common word and the normal sense of the word. Disambiguating different senses of normal English words (Mihalcea 2004; Navigli 2009; Chen et al. 2014) and resolving co-references (Yarowsky 1995; Navigli 2009) themselves are very challenging NLP problems, which are out of scope of the present paper.

Recently, Ye et al. (2016a) proposed a machine learning based approach, called S-NER, to recognize general software-specific entities, including APIs, in software engineering social content. S-NER's F1-score for API recognition is much lower than that of other types of software entities, such as programming languages and software standards. This is due to: 1) S-NER aims to recognize a broad category of software entities, making it very difficult to build a gazetteer with good coverage of APIs; 2) S-NER uses only basic context features, i.e., the word itself, word shape and word type of the surrounding words, and thus has limited toleration to context variations as these basic context features cannot capture the semantic similarity between different words. Moreover, there is no API linking performed in S-NER.

Considering the fact that an API is an entity in the context of software engineering, our approach for API linking belongs to the general category of entity linking. A community-curated list of publications on entity linking can be found at http://nlp.cs.rpi.edu/kbp/2014/elreading.html. Wikipedia has been widely used as the knowledge base to be linked to among these work. Contextual similarity based approaches are widely adopted (Shen et al. 2012; Mihalcea and Csomai 2007; Liu et al. 2013). We adapt the similarity based method used in Liu et al. (2013) to our specific problem of API linking in the context of Stack Overflow textual discussions.

4 The API extraction approach

In this section, we describe the two steps of our API extraction approach, i.e., API recognition and API linking. We name our **API REcognition And Linking** apporach as *APIReal*.

4.1 API recognition

Let T be a discussion thread, i.e., a question and all its tags, answers and comments, in Stack Overflow. A question or answer is referred to as a post in Stack Overflow. Let $S \in T$ be a natural language sentence with code snippets removed from a post.

Given a natural language sentence S, the API recognition step is to recognize all API mentions in S, as illustrated in the example in Fig. 1. Specially, we use the dynamically-typed programming language Python as a case study, and we want to extract tokens in a natural language sentence that refer to public modules, classes, methods or functions of certain libraries as API mentions. An API mention should be a single token rather than a span of tokens when the given sentence is tokenized properly, preserving the integrity of code-like tokens. API mentions can be of the following forms:

- *Standard API full name:* The formal full name of an API from the official API website, e.g., *pandas.DataFrame.apply* or *pandas.DataFrame.apply()* of the *Pandas* library;

- *Non-standard synonym:* Variants of a standard API name that are composed of commonly-seen library or class name synonyms, e.g., *pd.merge*, *pandas.df.apply*, or *pd.df.apply* in which *pandas* is written as *pd* and *DataFrame* is written as *df*;
- *Non-polysemous derivational form:* API mentions that can be partially case-insensitive matched to a standard API name or its non-standard synonym, e.g., *dataframe.apply, df.apply, .apply, or apply()*;
- *Polysemous common-word:* common-words that refer to the simple name of an API, such as Series (class) and apply (method) of *Pandas*, Figure (class) and draw (method) of *Matplotlib*, and Polynomial (class) and flatten (method) of *Numpy*.

In this work, we focus on tackling common-word polysemy and sentence-format variation issues in recognizing whether a token is an API mention of certain libraries. Figure 3 shows the main steps of *APIReal* for recognizing API mentions: 1) *APIReal* performs a *preprocess* step including code snippet removal, HTML tag clearing, and tokenization. 2) *APIReal* uses two unsupervised language models (i.e., class-based Brown clustering (Brown et al. 1992) and neural-network based word embedding (Turian et al. 2010; Mikolov et al. 2013b)) to learn *word representations* from unlabeled text and cluster semantically similar words. 3) *APIReal* constructs an *API inventory* based on the Brown clustering. 4) We select *training sentences* and *label* them. 5) *APIReal* trains a linear-chain *Conditional Random Field (CRF) model* based on the labeled sentences using orthographic features from tokens, compound word-representation features from the two different unsupervised language models, and gazetteer feature from the API inventory. 6) We also perform an iterative *self-training* process to alleviate the lack of labeled data for model training.

4.1.1 Text preprocessing

Motiviation In this study, we focus on recognizing API entities from natural language texts. Hence, given a Stack Overflow post, we need to perform a preprocessing step including code snippet removal, HTML tags cleaning, and tokenization.

Approach We remove large code snippets in <pre><code>, but keep short code elements in <code> in natural language sentences. We write a sentence parser to split the post text into sentences. For a natural language sentence, we use our software-specific tokenzier (Ye et al. 2016b) to tokenize the sentence. This tokenizer preserves the integrity of code-like tokens. For example, it tokenizes *pandas.DataFrame.apply()* as a single token, instead of

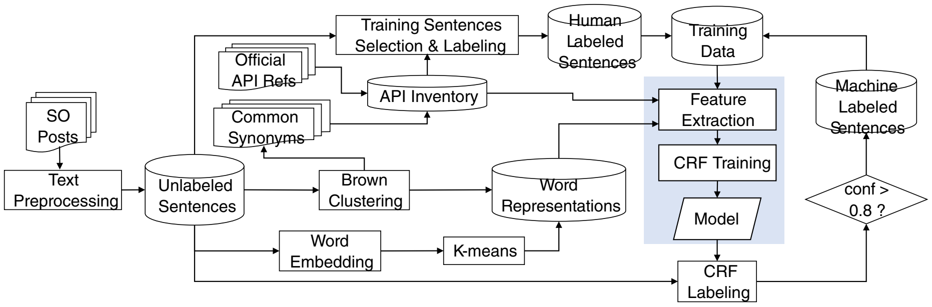


Fig. 3 API recognition of *APIReal*. The inputs are Stack Overflow sentences, and the output is a trained machine learning model that can recognize API mentions in the input sentences

a sequence of 7 tokens, i.e., *pandas . DataFrame . apply ( )*. For other texts (e.g., email), different preprocessing steps, sentence parser, and tokenizer may be needed.

4.1.2 Learning word representations

Motiviation In informal social discussions, both API mentions and sentence context vary greatly (see Tables 1 and 2). These variations result in out-of-vocabulary issue (Li and Sun 2014) for a machine learning model, i.e., variations that have not been seen in the training data. As we want only minimal effort to label training data, it is impractical to address the issue by manually labeling a huge amount of data. However, without the knowledge about variations of semantically similar words, the trained model will be very restricted to the examples that it sees in the training data. To address this dilemma, we propose to exploit unsupervised language models to learn word clusters from a large amount of unlabeled text. The resulting word clusters capture different but semantically similar words, based on which a common word representation can be produced to represent the words in a cluster. Word representations are then used as features to inform the model with variations that have not been seen in the training data.

Approach The state-of-the-art unsupervised language models include class-based Brown clustering (Brown et al. 1992; Liang 2005) and neural-network based word embedding (Mikolov et al. 2013b). Different language models make different assumptions about corpus properties to evaluate the semantic similarity of words. Empirical studies (Guo et al. 2014; Yu et al. 2013) show that Brown clustering and word embedding produce complementary view of the semantic similarity of words, and when combined together as compound features, they can significantly improve the performance of entity recognition techniques. Considering the informal and diverse nature of our text, we decide to use both Brown clustering and word embedding to learn word representations.

We assume that users are interested in extracting API mentions of a particular library (e.g., *Pandas* or *Numpy*). To learn unsupervised language models, we collect a large set of Stack Overflow posts that are tagged with the library, excluding those containing sentences selected as training data. The posts are preprocessed and split into sentences in the text preprocessing step. This produces a large set of unlabeled sentences. Unlike prior work (Guo et al. 2014; Yu et al. 2013; Ye et al. 2016a; Li and Sun 2014; Yao and Sun 2015), we do not convert words into lowercase. This is because many APIs have initial-capitalized name, e.g., the Series class of the *Pandas* library. We want language models to treat them as different words from their lowercase counterparts.

Given this set of unlabeled sentences, Brown Clustering (Brown et al. 1992) outputs a collection of word clusters. Each word belongs to one cluster. Words in the same cluster share the same bitstring representation. Previous studies (Yao and Sun 2015; Ye et al. 2016a; Li and Sun 2014) show that Brown clusters are useful to identify abbreviations and synonyms. Indeed, we exploit Brown clusters learned from unlabeled text to expand standard API names with commonly-seen name synonyms.

For neural-network based word embedding, we use a continuous skip-gram model (Mikolov et al. 2013a, b) to learn a vector representation (i.e., word embedding) for each word. Word embeddings have been shown to capture rich semantic and syntactic regularities of words (Mikolov et al. 2013b; Turian et al. 2010). However, studies (Guo et al. 2014; Wang and Manning 2013; Yu et al. 2013) show that it is inefficient to directly use the low-dimensional continuous word embeddings as features to a linear-chain CRF model for entity recognition, because the linear CRF theoretically performs well in high-dimensional discrete feature space. Therefore, following the treatment of prior work (Guo et al. 2014; Yu et al. 2013), we transform the word embeddings to a high-dimensional discrete representations leveraging the K-means clustering. Concretely, each word is treated as a single sample, and each K-means cluster is represented as the mean vector of the embeddings of words assigned to it. Similarities between words and clusters are measured by Euclidean distance. Similar to Guo et al. (2014), we set K to a set of values (e.g., 500, 1000, 1500, 2000, 2500) to obtain a set of K-means clusters. After K-means clustering, each word is represented as the ID of the cluster in which the word belongs to, i.e., a one-shot K-dimensional vector in which the ith dimension is set to 1 if the word belongs to the ith cluster and all other dimensions are set to 0.

Word representations obtained from the Brown clusters and the word embedding clusters are used as features to the CRF model. This helps the CRF model tolerate semantically similar API-mentions and sentence-context variations, and thus alleviate the out-of-vocabulary issue.

4.1.3 Constructing API inventory

Motiviation A gazetteer of known entities is often compiled for NER and WSD tasks. Partial name match of gazetteer entities is commonly used as an important feature for the CRF training, which has been shown to improve the performance of the trained model. However, our previous work on software-specific NER shows that a gazetteer of standard API names contributes only marginally to the NER performance. This is because of the wide presence of non-standard API synonyms and their derivational forms in informal natural language texts (see Table 1 for examples). Therefore, we construct an API inventory for a library that contains not only standard API names but also commonly-seen synonyms of API mentions.

Approach Our approach adopts and combines the best practices for API inventory construction from API recognition and linking works (Bacchelli et al. 2010; Dagenais and Robillard 2012; Rigby and Robillard 2013) as well as from entity extraction works in general domain (Mihalcea 2004; Navigli 2009; Chen et al. 2014). Given a library, we first crawl a list of standard API names from the library's official website. For example, for the Pandas library, the list of standard API names includes *pandas.DataFrame*, *pandas.DataFrame.apply*, etc. Following the treatment of prior NER (Liu et al. 2011; Li and Sun 2014) and WSD (Navigli 2009; Mihalcea 2004) work, we remove extremely common English words from the inventory, such as *data*, *all*, because most of mentions of these extremely common English words are not API mentions.

Then, we examine the Brown clusters that contain the standard API names and their derivational forms, from which we can easily observe tokens that are semantically similar to the standard API names and their derivational forms, but written in different synonym forms. We infer commonly-seen synonyms of API mentions from these tokens, e.g., *pandas* written as *pd*, *DataFrame* written as *df*.

In our study, we observe that synonyms of library and class/module names are common, while we rarely see synonyms of method/function names (except for some misspellings). Therefore, we infer synonyms of standard API names using a simple combination of the observed library/class/module name synonyms. As our goal is not to compile a complete list of API synonyms, the analysis of commonly-seen synonyms does not require much effort. According to our experience, constructing the API inventory for a library requires only 2-3 hours, if the developer is familiar with web scraping and Brown clustering techniques.

The API inventory serves two purposes: 1) partial match of API names or synonyms in the inventory is used as a feature for the CRF; 2) ensuring that training data and test data reach a good coverage of polysemous and derivational forms of library APIs.

4.1.4 Training sentences selection and labeling

Motiviation The quality and amount of human labeled data for model training are essential to the performance of a machine learning system. However, there have been no dedicated efforts for labeling APIs in natural language sentences for tackling common-word polysemy issue in the task of API extraction. To train an effective machine learning model for distinguishing the API sense and the normal sense of common words, the labeled data must contain not only API mentions with distinct orthographic features but also sufficient polysemous common-word API mentions. Similar treatment has been adopted in word sense disambiguation research (Chen et al. 2014; Mihalcea 2004).

Approach In our work, we select training sentences that mention APIs of a particular library (e.g., *Pandas* in our evaluation), based on the API inventory of the library. However, the trained machine learning model is not limited to extracting API mentions of this particular library. Instead, it can robustly extract API mentions of very different libraries (e.g., *Numpy*, *Matplotlib*).

Inspired by the ambiguous location name extraction work (Li and Sun 2014) and the mobile phone name extraction work (Yao and Sun 2015), we propose to generate training data with minimal human labeling effort as follows. We manually split the APIs in the API inventory into two subsets based on whether an API's *simple name* has distinct orthographic features and whether the simple API name can be found in a general English dictionary. The simple name of an API in the *non-polysemous set* must have unambiguous orthographic features, for example, camel case *MultiIndex*, underscore *read_csv*, or must not be found in a general English dictionary, for example, *swaplevel*, *searchsorted*. In contrast, the simple name of an API in the *polysemous set* does not have distinct orthographic features and the simple name is a general English word, for example, *series*, *apply* and *merge*. Although the qualified name of an API always has distinct orthographic features, such as *pandas.series*, *apply()*, the simple name can be polysemous.

We select Stack Overflow sentences for labeling as follows. First, we randomly select 300 sentences from the posts that are tagged with the particular library. Each of these 300 sentences must contain tokens that exactly match the standard name of at least one API in the non-polysemous set, but must not contain tokens that match the simple name of the APIs in the polysemous set. Different sentences may mention the same APIs in the non-polysemous set. For these 300 sentences, we do not need to manually label the sentences. Those tokens that exactly match the standard name of the non-polysemous APIs can be automatically labeled as API mentions. Second, we randomly select sentences that contain tokens that match the simple name of at least one API in the polysemous set. These sentences contain tokens that can not only be API mentions but also be common words. Therefore, we must manually examine the selected sentences and label API mentions (if any) in the sentences. The selecting and labeling continues until we collect sentences that contain at least 200 mentions of the APIs in the polysemous set.

The selected sentences constitute the set of human labeled data for model training. This initial set of training data will be expanded using self-training, as discussed in the next step.

4.1.5 CRF-based classifier

Motivation Given a token in a natural language sentence, *APIReal* determines whether the token is an API mention or a normal word using a linear-chain Conditional Random Fields (CRF) (Lafferty et al. 2001). The CRF classifier is the state-of-the-art model for sequential labeling, which is particularly strong at learning contextual features. In our work, the CRF classifier is trained using a small set of human labeled sentences and a large set of machine labeled sentences obtained through self-training. After training, the classifier can be used to label the tokens of unlabeled sentences as API mentions or normal words.

Approach In this work, we design three kinds of features for the CRF classifier: *orthographic features* of the current token (word) and its surrounding tokens, *word-representation features* of the current token and its surrounding tokens, and *gazetteer features* based on the API inventory. To illustrate our feature design, we use the following notations: w_i denotes the current token. w_{i+k} denotes the next kth token to the current token, e.g., w_{i+1} is the next token to the current token. w_{i-k} denotes the previous kth token to the current token.

- **Orthographic features**. This set of features include: 1) *exact token*, including the current token w_i, the surrounding tokens of the current token in the context window [-2, 2], the bigrams $w_{i+k}w_{i+k+1}$ $(-2 \leq k \leq 1)$ in the context window [-2, 2], i.e., $w_{i-2}w_{i-1}$, $w_{i-1}w_i$, $w_i w_{i+1}$, $w_{i+1}w_{i+2}$; 2) *word shape* of the current token w_i and its surrounding tokens in the context window [-2, 2], including whether the token contains dot(s) and/or underscore, and whether the token is suffixed with a pair of round brackets; 3) *word type* of the current token w_i and its surrounding tokens in the context window [-2, 2], including type indicates if the token is all-capitalized or first-letter-capitalized, if it is made of all-symbol, all-letter, all-digit, a mixture of symbol and letter, etc.
- **Word-representation features**. For K-means clusters of word embeddings, each word in the corpus is assigned with a cluster ID. We denote the cluster ID of the current word as c_i. Following the pioneer work of utilizing compound cluster features (Guo et al. 2014; Yu et al. 2013), our word-embedding-cluster features are: 1) the cluster ID of the current word and its surrounding words in the context window [-2, 2]; 2) the bigrams of the cluster ID of the words within the context window, i.e., $c_{i+k}c_{i+k+1}$ $(-2 \leq k \leq 1)$; 3) the bigram of the cluster ID of the previous word and the next word, i.e., $c_{i-1}c_{i+1}$. For Brown clustering, each word is represented as a bitstring. Our Brown-cluster features are: 1) the bitstring of the current word and its surrounding words in the context window [-2, 2]; 2) the prefixes of the bitstring of the current word and its surrounding words in [-2, 2]. The prefix lengths we use in this work are {2, 4, 6, 8, ..., 14}.
- **Gazetteer features**. We use the API inventory as the gazetteer. Each standard API name or name synonym is an entry of the gazetteer. We perform string matching to the entries of the gazetteer, and use the matching result as our gazetteer feature. In particular, given a token w, we first remove the "()" if w is suffixed with "()". The resulting word, denoted as w_nb, is then matched to the gazetteer using the following criteria: 1) if w_nb contains no dot or w_nb ends with a dot, we perform exact matching to the gazetteer entries; 2) if w_nb is prefixed with a dot, we consider it as a match if any entries end with w_nb; 3) if w_nb contains dot in the middle, we consider it as a match if any entries begin with w_nb or partially match to $.w\_nb.$ (i.e., with a prefix dot and a suffix dot). We have the third rule because, if users write "e.g", a simple partial string matching will match the token to the API name like "pandas.cor**e.g**roupby.GroupBy.transform", which is not desired.

4.1.6 Iterative self-training

Motiviation The quality of a machine learning model relies on the sufficient, high-quality training data. However, we only manually label a small set of training data in this work. Although using unsupervised word representations alleviates the out-of-vocabulary issue, to further alleviate the lack of training data, we propose to use an iterative self-training mechanism (Wu et al. 2009), through which high-confidence machine labeled sentences will be added to the training dataset to retrain the model incrementally. This self-training process will expose the model to much more sentence variations that have not been covered by human labeled data.

Approach Algorithm 1 outlines the self-training process. The algorithm first trains a CRF classifier using the small set of human labeled data. Then, for each unlabeled sentence S, the algorithm uses the current CRF classifier to label the sentence and obtains a machine labeled sentence $S_{labeled}$ and the confidence $conf$ of the labeling result (lines 4-5). If the labeling confidence is above the user-specified threshold α, the machine labeled sentence is added to the set of labeled training sentences (lines 6-9). Once more than N machine labeled sentences are added, the algorithm retrains the CRF classifier with the larger set of labeled sentences (including both human labeled and machine labeled) (lines 10-14). The new CRF classifier will be used to label the rest of the unlabeled sentences. The process continues until all unlabeled sentences are processed or the maximum number iterations has been done.

Algorithm 1 Self-training the CRF-based Classifier

```
Data: A stream of unlabeled sentences $unlabelsents$;
      A set of labeled training sentences $te$;
Result: The CRF classifier $l$
1  $te \leftarrow$ human labeled sentences;
2  $l = train(te)$;
3  for $S \in unlabeledsents$ && $iterations < M$ do
4      $\mathbf{S} = feature\_extractor(S)$;
5      $(S_{labeled}, conf) = crf\_label(l, \mathbf{S})$;
6      if $conf > \alpha$ then
7          $te \leftarrow te \cup \{S_{labeled}\}$;
8          $n = n + 1$;
9      end
10     if $n > N$ then
11         $l = train(te)$;
12         $iterations + +$;
13         $n = 0$;
14     end
15 end
```

4.2 API linking

After we perform API recognition over a piece of natural language texts, our next task in this paper is to disambiguate the correctly recognized API mentions to its unique formal form (i.e., fully qualified name), as illustrated in the example in Fig. 2. API recognition tells whether a given token is an API or not, while disambiguating which specific API an API mention refers to is the job of API linking. To perform API linking, we need an API knowledge base for linking. Each entry of the API knowledge base is the target to be linked to Subramanian et al. (2014), and each mention of the recognized APIs in the text is the source to link from. Figure 4 shows the main steps of our rule-based approach for API linking, which consists of three components: mention-mention similarity, scope filter, and mention-entry similarity. The input is the content of a discussion thread with API mentions recognized, which is pipelined through the three components. The output is a ranked list of APIs to be linked.

Note that not all API mentions are ambiguous thus require linking. For example, if an API is mentioned with its fully qualified name, there should be only one linking candidate in the knowledge base. It is also easy to perform linking if an API's simple name is unique. Thus, to evaluate if an API linking approach works, we have to focus on those API mentions that are highly ambiguous, i.e., those with multiple mappings in the API knowledge base. An API is considered correctly linked if and only if both the recognition and linking are correct.

Knowledge base population Unlike existing entity linking work that uses Wikipedia as the knowledge base for general domains, there has been no publicly-available knowledge base for our specific API linking task. Therefore, we construct a knowledge base consisting the information of our studied Python APIs. Such knowledge base has also been named as *code element index* in the software engineering community (Rigby and Robillard 2013).

We crawl API information for each studied library from their official site. Inspired by the API information crawled in other works (Bacchelli et al. 2010; Dagenais and Robillard 2012; Rigby and Robillard 2013), each entry (or record) in our knowledge base contains 7 fields:

1. ID, i.e., the unique ID of an entry.
2. name. We store the fully qualified API as it appears in the official API documentation in this field.
3. URL, which is the unique URL address for the API.
4. library, which is the library the API belongs to (Pandas, Numpy, Matplotlib or Python Standard Library).
5. class, which represents the declaring class of the API if applicable. If the API is a method or function, then there has to be a class or a module hosting the API according to the naming convention of Python. If the API itself is already a class or module, then this field is populated with ‘N/A’.
6. type, e.g., if the API is a method or a class. In our case, we consider the API type of a Python module or class as *class*, and we consider the type of a Python function and method as *method*.

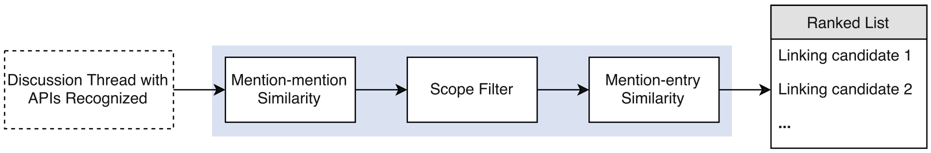


Fig. 4 API linking of *APIReal*. The input is the content of the discussion thread with API mentions recognized, and our API linker links an API mention to its fully qualified form in the official documentation

7. description. This field consists of the textual descriptions of the API from the official API documentation.

Linking API mentions to knowledge base entries Once we recognize an API mention, we match it against the "name" field of our knowledge base. Note that the "name" field stores fully qualified API name, while the API mentions on Stack Overflow are mostly unqualified. Hence, we design regular expressions to perform partial matching. We add a dot as the prefix if the mention is the simple name of the API. For example, if the recognized API mention is written as "apply", our regular expression for partial matching will be "*.apply". We also perform lightweight normalization to API mentions that are written in derivational forms, using the name variations observed when we constructed the API inventory. For example, "df.apply" will be normalized as "dataframe.apply" before we match it to the knowledge base entries. The matching to the knowledge base can potentially give us a list of linking candidates. We then use the three components i.e., mention-mention similarity, scope filter, and mention-entry similarity, to determine which linking candidate is the correct one to be linked to. The three components are executed in sequence. The mention-mention similarity component is determinative to the output once it is activated. The scope filter is to help reduce the number of linking candidates. Finally, the mention-entry similarity component is used to rank the linking candidates.

- **Mention-mention similarity.** This component checks if there exists similar API mentions in the context that have been linked or is relatively unambiguous. For a given recognized API mention, we examine the global context within the discussion thread by checking: 1) if the same API mention has been manually added with a URL link by Stack Overflow users; and 2) if a relatively formal version of the same API mention can be found. For example, in this Stack Overflow question http://stackoverflow.com/questions/35782929, "groupby" is an Pandas API. One of the "groupby" mentions has been added with a URL link to the official webpage of "groupby". In this case, if some other "groupby" mentions without manual links are recognized as APIs by our API recognition, we link these "groupby" mentions to the same URL.
- **Scope filter.** This component consists of a set of heuristics leveraging the global context of the current discussion thread to narrow down the scope of API linking. We first check if the declaring library of a possible candidate can be found in the tags of the question or title of the question. This is done through simple lowercase string match. Stack Overflow tags help to sort questions into specific, well-defined categories so that we can have a general idea about what the question is asking about. One Stack Overflow question has at least one tag. Similarly, the question title summarizes the topic of the question. With the title and tags information, we can potentially eliminate non-relevant candidates from other libraries.

  Although we are performing API linking in text rather than the code blocks on Stack Overflow, we utilize the code blocks posted in the question or answer to reduce the number of linking candidates. Particularly, we use code blocks to find the potential declaring class (type) of the API mentioned in the nearby text, so as to resolve ambiguities for method or function names. Stack Overflow uses the Google Code Prettify Library[1] to perform lightweight syntax highlighting of code blocks in posts: code elements like keywords, types (classes), literal value, comments, etc., are augmented with

[1] https://github.com/google/code-prettify

<span> HTML tags of different classes. This provides a good source of information to design our scope filters, because the potential declaring classes can be obtained easily by selecting span elements whose class attribute is *typ*. Extracting the type information from the code blocks can be useful to cases where the post contains few textual descriptions. We assume that a method is likely to be discussed together with its declaring class rather than other classes. Therefore, once a class can be found from the code blocks, we can use the class to help disambiguate method names recognized in discussion thread. We further use the recognized API mentions that are classes (i.e., the recognized API is a class) for a similar job. Recall that in our knowledge base, for each API, we stored its type either as a class or a method. After we recognize an API mention, we further check its type by querying the knowledge base. If the type is class, we use it to disambiguate method mentions that are written in their simple names.

- **Mention-entry similarity.** Although the scope filter can potentially reduce the number of potential candidates, it may not be able to reduce the number of linking candidates to only 1, i.e., finding the specific fully qualified API name, because 1) the tags and title matching only works at declaring library level; 2) we may not always find type/class from code blocks. Similarly, the mention-mention similarity component is not applicable in cases where there is no related reference URL or formal version of API mentions found in the discussion. To overcome these shortcomings, we compute the mention-entry similarity, which is the textual similarity between a discussion thread and the documentation of an entry of the knowledge base. Specifically, after we apply the scope filter, we compare the content of the current discussion thread with the content of each of the linking candidates. We represent the content of the discussion thread and the content of the documentation of the entry as term frequency–inverse document frequency (TF-IDF) vectors. In our context, TF is the number of times a token appear in the current document while IDF is the inverse of the number of documents a token appears across the collection of documents. Here the collection of documents refers to the current discussion thread plus all the descriptions of the linking candidates (one description is one document). We use logarithmic scaling because it is the standard way to reduce the distorting effect of code elements that are redundantly repeated in a single document (Rigby and Robillard 2013). We then calculate the textual similarity between the TF-IDF vector and each of the TF-IDF vectors of the linking candidates using cosine similarity, based on which we rank the linking candidates.

5 Experimental setup

This section describes tools we use to implement *APIReal*, studied libraries, model training settings, human labeling of test dataset, evaluation metrics, and the baseline methods we use to compare *APIReal*.

5.1 Implementation and replication package

We implement web crawlers using Scrapy[2] to crawl official API names. For the implementation of the linear CRF, we use CRFsuite,[3] a popular CRF toolkit for sequential labeling. For

[2]Scrapy, http://scrapy.org/

[3]CRFSuite, http://www.chokkan.org/software/crfsuite/

Table 4 General information of the studied libraries

Library	Version	#Questions	Attribute	#APIs	#Polysemous API	%
Pandas	0.18.0	22,226	Panel data analysis	774	426	55.04%
Matplotlib	1.5.1	16,480	2D plotting	3877	622	16.04%
Numpy	1.10.1	24,390	Scientific computing	2217	917	41.36%

Brown Clustering, we use Liang's implementation.[4] We learn continuous word embeddings using word2vec,[5] which contains an efficient open-source implementation of the skip-gram model (Mikolov et al. 2013b). We use the K-means implementation from Sofia-ML [6] to perform K-means clustering of the continuous word embeddings.

A replication package of our study can be downloaded at https://goo.gl/oeU7k8. This replication package contains the implementation of our API recognition and linking method. We provide training and testing data in format of CoNLL[7] file, which is widely used in Natural Language Processing (NLP). For API recognition, users can convert a text file into a CoNLL file using the python script *texttoconll.py*. Then, users can extract features based on CoNLL files using the python script *enner.py*. Finally, users can use CRFsuite to learn a CRF model. For the API Linking, users can use the python script *apilink.py*. We also provide the API inventory used in our study as a MySQL dump and the results of word representations using Brown cluster and word embedding.

5.2 Studied libraries

For API recognition, the key challenge is to distinguish the API sense of a common word and the normal sense of the word in a natural language sentence. To evaluate whether *APIReal* achieves this objective, we need to choose libraries that often use common words as API names. To this end, we choose three Python libraries, i.e., *Pandas*, *Numpy*, and *Matplotlib*. Table 4 summarizes the information of the three libraries, including the number of Stack Overflow questions that are tagged with the corresponding library tag. We construct the API inventory for the three libraries. APIs in the inventory are then split into a non-polysemous set and a polysemous set. *Pandas*, *Numpy* and *Matplotlib* have 55.04%, 16.04% and 41.36% APIs whose simple name is polysemous common word, respectively.

For API linking, considering the scope of the case study in this paper, we construct a knowledge base consisting of the API information of the above three libraries and the Python Standard Library. The information is crawled from the data sources listed in Table 5. In total, our knowledge base is populated with 13713 entries.

5.3 Model training and testing for the learning-based recognition

Training The sentence selection and labeling process has been described in Section 4.1. The labeling results are cross-checked by the first author and the fourth author to reach agreements. We use Fleiss Kappa (Fleiss 1971) to measure the agreement between two

[4]Brown Clustering, https://github.com/percyliang/brown-cluster

[5]Word2vec, https://code.google.com/archive/p/word2vec/

[6]Sofia-ML, https://code.google.com/archive/p/sofia-ml/

[7]http://www.signll.org/conll/

Table 5 Data source for knowledge base population

Library	Source
Pandas	http://pandas.pydata.org/pandas-docs/stable/api.html
Matplotlib	http://matplotlib.org/api/index.html
Numpy	http://docs.scipy.org/doc/numpy-1.10.1/genindex.html
Python standard	https://docs.python.org/2/library/index.html

annotators. The Kappa value between two annotators is equal to 0.893, which indicates an almost perfect agreement between the annotators (see the interpretations of Kappa values in Table 6). This level of agreement is because most of APIs are easy to be determined by human. The disagreements between the two annotators are mainly due to misunderstandings of APIs. Such disagreements can be resolved easily by resorting to the API documents.

Unlabeled sentences used to learn Brown clusters and word embeddings include all the sentences from Stack Overflow that are tagged with `pandas`, `numpy` and `matplotlib`. For Brown clustering, we ignore the words that appear fewer than 3 times in the unlabeled sentences, and the number of Brown clusters is set to 500. For word-embedding clusters using K-means, we follow the settings of Guo et al. (2014), i.e., we set K to 500, 1000, 1500, 2000, 2500 to get 5 clustering results. These 5 word-embedding clusters and the Brown clusters are used as word-representation features to the CRF.

For the self-training process, we randomly select unlabeled sentences using the API inventory of the *Pandas* library, and feed these sentences as a stream of unlabeled sentences to Algorithm 1. We iterate the self-training 10 times (i.e., $M = 10$). We follow the empirical parameter settings of prior work (Mihalcea 2004; Liao and Veeramachaneni 2009; Liu et al. 2011). The threshold of confidence score for adding a machine labeled sentence into the training set is 0.8, i.e., α at line 6 of Algorithm 1. With this high threshold, machine labeled sentences will not introduce much noise to the model. Meanwhile, it is not too strict so that the self-training can expand the model with unseen examples that are different from the training examples that are already in the training set. We set N to 500, i.e., once 500 high-confidence machine labeled sentences are added into the training set, we re-train the model.

Testing For each of the three studied library, i.e., Pandas, Numpy and Matplotlib, we randomly select and label natural language sentences from the Stack Overflow posts and comments that are tagged with the corresponding tag. We stop labeling once we obtain at least 150 sentences, each of which must contain at least one mention of an API in the API

Table 6 The interpretations for Kappa values

Kappa value	Interpretation
< 0	poor agreement
[0.01, 0.20]	slight agreement
[0.21, 0.40]	fair agreement
[0.41, 0.60]	moderate agreement
[0.61, 0.80]	substantial agreement
[0.81, 1.00]	almost perfect agreement

inventory of the library. The mention can be standard name, non-standard synonym, or non-polysemous derivational form of the API. Meanwhile, our testing data must also contain at least 150 sentences, each of which must contain at least one mention of a polysemous API by its simple name. The labeling results are cross-checked by the first and third author to reach agreement on the labels.

In the end, our testing dataset has 3,389 sentences containing 65,857 tokens. Among these 3,389 sentences, 903 sentences (26.6%) contain at least one API mention. Table 7 summarizes the statistics of different forms of API mentions. In total, the testing data contains 1,205 API mentions for the three libraries, which refer to 33.9%, 36.1% and 30% of the APIs of the respective library. Among the 1,205 total API mentions, 44% of API mentions (531 times) in our testing data are polysemous common-word mentions.

6 Experiment results and analysis

We now report experiment results and analyze our findings.

6.1 API recognition

6.1.1 Baselines

We compare the learning-based API recognition approach of *APIReal* with three state-of-the-art methods for fine-grained API recognition from natural language texts.

– ***LightRegExp* - Lightweight regular expressions**. We implement lightweight regular expressions used in Miler (Bacchelli et al. 2010). Specifically, Miler supports dictionary look-up combined with lightweight regular expressions to extract APIs from emails. Regular expressions are defined based on language convention and one punctuation rule. Same to Miler, we perform dictionary look-up in our API inventory and devise lightweight regular expressions based on Python's language conventions (e.g., check the existence of dot and underscore). We use the same punctuation checking rule as Miler, which checks if a token is surrounded by punctuations (please refer to Subsection "Punctuation" in Section 4 of Miler (Bacchelli et al. 2010) for details).
– ***CodeAnnotationRegExp* - Code annotation enhanced regular expressions**. We combine regular expression baseline with code annotations (e.g., HTML tags $< code >< /code >$, $< a >< /a >$) to extract API entities on Stack Overflow. Some previous studies (e.g., Parnin et al. 2012 and Linares-Vásquez et al. 2014) have shown that these code annotations based on HTML tags can help users identify and link API entities. In this baseline, the token that is annotated with <code> can be considered as an "island"

Table 7 Statistics of API mentions in testing dataset

Library	#API Mentions			
	Standard/deriv[1]	Synonym/deriv	Polysemy	Total
Pandas	167	59	182	408
Matplotlib	184	62	189	435
Numpy	88	114	160	362
Total	439	235	531	1,205

[1]deriv = derivational form

(i.e., an API mention), which is the island parsing idea from the approach of Rigby and Robillard (2013). However, our baseline approach does not consider immediate and local context for sophisticated context analysis, which is used in the approach of (Rigby and Robillard 2013).

– ***MLNER* - Machine-learning based NER**. We use the software-specific entity recognition tool (S-NER) proposed in our earlier work (Ye et al. 2016a) to recognize the API mentions in our testing data. For fair comparison, we use the same set of features used in Ye et al. (2016a), and re-train the model of S-NER with the same set of human labeled sentences for training the CRF model of this work.

6.1.2 Results

Metrics We use precision, recall, and F1-score to evaluate the performance of an API recognition method. Precision measures what percentage the recognized APIs are correct; recall measures what percentage the API mentions in the testing dataset are recognized correctly by a method; and F1 is the harmonic mean of precision and recall.

Overall performance Table 8 shows the performance differences of the three evaluation metrics for using the three baseline methods and our *APIReal* to extract all API mentions in the testing dataset. The API recognition method of *APIReal* outperforms all the baseline methods. It achieves the best and balanced precision and recall, and the F1-score is 0.876.

– *Performance Analysis for LightRegExp:* we observe almost the same performance result as that of Miler (Bacchelli et al. 2010) for extracting API mentions of the C library *Augeas*. Miler's performance for extracting mentions of the *Augeas*'s APIs from developer emails: precision 0.15, recall 0.64 and F1-score 0.24. In our experiment, the method *LightRegExp* (i.e., dictionary look-up and lightweight regular expressions) achieves precision 0.125, recall 0.723 and F1-score 0.213. This is because both the C library *Augeas* and the three Python libraries used in this experiment define many common-word APIs, which creates common-word polysemy issue once mentioned by their simple name in the text. Miler's approach resolves the issue by aggressively labeling common-word tokens as APIs, and thus achieves very low precision but good recall. If a conservative strategy were adopted, the result would go the opposite, i.e., improved precision but degraded recall. Overall, the method *LightRegExp* cannot properly address common-word polysemy issue.
– *Performance Analysis for CodeAnnotationRegExp:* This baseline, which combines code-annotation enhanced regular expressions with code annotations, proves to be more useful and reliable for API extraction from informal text. *CodeAnnotationRegExp* achieves balanced precision and recall, and the F1-score is 3 times higher than that of the first baseline method *LightRegExp*. However, it still misses about 38% of the API mentions and about 37% of the extracted API mentions are not true API mentions. This

Table 8 Performance differences between different API recognition methods

Method	Precision	Recall	F1-score
LightRegExp	0.125	0.723	0.213
CodeAnnotationRegExp	0.633	0.624	0.628
MLNER	0.825	0.678	0.744
APIReal	0.879	0.872	0.876

baseline approach especially falls short to extract API mentions when users forget to annotate the API mentions, such as the mention of the *series* class and the *apply* mention in Fig. 1, which is common in Stack Overflow discussions. This might be because our baseline *CodeAnnotationRegExp* does not consider immediate and local context, which is used in the approach proposed by Rigby and Robillard (2013).

- *Performance Analysis for MLNER:* This baseline method *MLNER*, i.e., machine-learning based software-specific named entity recognition, achieves significantly higher precision and a moderate improvement on recall, compared with the second baseline method *CodeAnnotationRegExp*. *APIReal* can improve the precision even further, and meanwhile significantly improve the recall. The improvement on recall over *MLNER* can be attributed to the use of unsupervised word representations as compound semantic context features in our method. In contrast, *MLNER* uses only simple orthographic context features, and thus its model puts more weight on the orthographic features and word representations of the current word, and less on context features. As a result, the improvement of *MLNER* on recall is moderate. Other new features introduced in *APIReal*, such as commonly-seen synonyms in API inventory and self-training, also contribute to boosting up precision and recall, compared with our previous machine-learning based method (Ye et al. 2016a).

Individual libraries Table 9 shows the comparison of the API extraction performance of different methods for the three studied libraries, respectively. Similar observations can be made as the comparison of the overall performance.

An interesting observation is the performance improvement of the baseline *MLNER* and *APIReal* (i.e., two different machine-learning based methods) *across* libraries. The baseline *MLNER* performs the best on extracting mentions of *Pandas*'s APIs (F1-score 0.823), but the performance drops significantly for *Numpy*'s and *Matplotlib*'s APIs (F1-score 0.749 and 0.629 respectively). Similarly, *APIReal* also performs the best for *Pandas*'s APIs (F1-score 0.901), but the performance of *APIReal* drops only slightly for *Numpy*'s and *Matplotlib*'s APIs (F1-score 0.860 and 0.867 respectively).

Recall that we train the model of the baseline *MLNER* using sentences mentioning some *Pandas*'s APIs. This model captures the knowledge about orthographic features and semantic representations of *Pandas*'s APIs. Although *Pandas*'s APIs mentioned in the testing dataset are different from those mentioned in the training data, they are all from the same library, share similar orthographic features, and serve the overall similar semantics. As a result, the knowledge learned from some *Pandas*'s APIs can help extract mentions of other *Pandas*'s APIs in the testing dataset. However, this knowledge cannot be transfered to other libraries that have different orthographic features and support different functionalities.

Table 9 API recognition performance for the 3 studied libraries

Method	Pandas			Matplotlib			Numpy		
	Prec.	Rec.	F1	Prec.	Rec.	F1	Prec.	Rec.	F1
LightRegExp	0.153	0.791	0.257	0.107	0.689	0.180	0.111	0.675	0.191
CodeAnnotationRegExp	0.640	0.615	0.627	0.611	0.622	0.617	0.617	0.645	0.631
MLNER	0.858	0.791	0.823	0.779	0.527	0.629	0.795	0.705	0.747
APIReal	0.913	0.889	0.901	0.856	0.879	0.867	0.847	0.873	0.860

Therefore, the performance of the baseline *MLNER* drops significantly, especially for *Matplotlib* which is more distant from *Pandas* than *Numpy*.

In addition to orthographic features and semantic representations of API mentions, *APIReal* exploits two new features, i.e., commonly-seen name synonyms and semantic representations of surrounding context of API mentions. Both features are derived from unsupervised language models learned from abundant unlabeled text. The knowledge about common synonyms and semantics of surrounding context, albeit obtained through unsupervised learning, makes *APIReal* more robust than the baseline *MLNER* for extracting mentions of *Numpy*'s and *Matplotlib*'s APIs.

Feature ablation We ablate one kind of feature(s) at a time from our full feature set and study the impact of different kinds of features on the API extraction performance. Table 10 reports the experiment results on precision, recall and F1-score. For orthographic features ablation, we ablate word shape and word type features, but retain the current word itself and its surrounding words as feature. Without orthographic features, the F1-score drops slightly to 0.858. Without word-representation features for the current token and its surrounding tokens, the F1-score drops to 0.828. Without gazetteer feature, the F1-score decreases to 0.801. And without self-training, the CRF model trained using only human labeled sentences achieves a F1-score of 0.85.

This result implies that the performance of *APIReal* is contributed by the combined action of all its features. However, features from unsupervised word representations and API inventory have a larger impact on the performance than orthographic features of tokens. Without a particular kind of features, *APIReal* still outperforms the best baseline method (i.e., *MLNER*). However, when ablating both features from word representations and API inventory, i.e., only orthographic features are retained, the performance of *APIReal* deteriorates significantly, and becomes worse than two baselines, i.e., CodeAnnotationRegExp and *MLNER*. This indicates the importance of word-representation and gazetteer features that *APIReal* introduces. Furthermore, the results also show that with unsupervised language models, *APIReal* can already achieve good performance. With small effort to construct the API inventory, the performance can be further improved. If the optimal performance is desired, users may also consider spending some manual efforts to annotate a small set of sentences mentioning APIs of the target library and retrain the model through the self-training process.

6.2 API linking

We select recognized API mentions from Stack Overflow questions that are tagged with either Pandas, Numpy or Matplotlib. Note these questions may contain more than 1 tag.

Table 10 The impact of one kind of feature(s)

	Precision	Recall	F1-score
Full-features	0.879	0.872	0.876
w/o orthographic features	0.842	0.871	0.858
w/o word representations	0.816	0.849	0.828
w/o gazetteer features	0.837	0.761	0.801
w/o word representations&gazetteer features	0.745	0.447	0.559
w/o self-training	0.861	0.839	0.850

To evaluate the linking efficiency, all the API mentions chosen are method names, because method names have generally more number of mappings in the knowledge base compared to class names, except for cases where a method is written in a relatively formal form, such as “dataframe.apply”, which only has one mapping in the knowledge base when we perform partial matching. In total, we select 120 API mentions, 60 of them are from Pandas questions, 30 from Matplotlib questions and 30 from Numpy questions. Note that the size of our ground truth dataset is similar to that of Baker (Subramanian et al. 2014). On average, each of the selected API mentions has 7.6 linking candidates in the constructed knowledge base. We create the ground truth data by manually inspecting the context of each API mention we selected to identify its correct linking result. We then compare the linking results given by the API linking approach of *APIReal* with our ground truth data labelings.

The experimental results are shown in Table 11. We mark it as a true positive (TP) when the linked result of *APIReal* is the same as the ground truth labeling. It is a false positive (FP) when the linked result does not match the labeling. A false negative (FN) is for the case when there is no linking result returned to the API mention but we would have expected to see a linking results. Precision is calculated as TP/(TP+FP), recall is calculated as TP/(TP+FN), and F1 is the harmonic mean of precision and recall. We can see that the API linking approach of *APIReal* is able to achieve an overall F1-score of 0.884. Using *APIReal*, the highest F1-score happens for the case of the Pandas Library (F1-score is 0.909). We observe similar amount of FP and FN cases for each of the 3 studied libraries.

Furthermore, we also find that TF-IDF based cosine similarity does not always produce the highest similarity score for the correct API description. Sometimes it can only be useful to shrink the pool of candidates instead of picking the correct candidate directly like the mention-mention similarity component, which is similar to the scope filter component. For example, in this Stack Overflow question http://stackoverflow.com/questions/32350288/, for the linking of the recognized API *errorbar*, the mention-mention similarity component is not activated, and the scope filter is only able to shrink the linking scope to APIs of the Matplotlib Library, *APIReal* links it to the wrong entry in the knowledge base (*matplotlib.axes.Axes.errorbar*) instead of the correct one (*matplotlib.pyplot.errorbar*) due to similar TF-IDF scores. To overcome the limitation of TF-IDF on measuring mention-mention and mention-entry similarity, we will consider more advanced word representation approaches (e.g., topic distribution vector based on LDA topic model, word embedding, etc.) in the future work to improve the linking quality.

7 Scalability of *APIReal*

For API recognition, our experiments demonstrate the generality of our approach for extracting API mentions of three very different Python libraries from Stack Overflow

Table 11 Experimental results of the API linking approach of *APIReal*

Library	#APIs	TP	FP	FN	Precision	Recall	F1
Pandas	60	50	5	5	0.909	0.909	0.909
Matplotlib	30	22	3	5	0.880	0.815	0.846
Numpy	30	23	4	3	0.852	0.885	0.868
Total	120	95	12	13	0.888	0.880	0.884

sentences. To expand *APIReal* to a new library, users need to prepare two kinds of information, i.e., unsupervised language models and API inventory. To learn unsupervised language models, users only need to collect a large corpus of unlabeled text, for example, Stack Overflow posts that are tagged with the library name. Then, the learning is completely unsupervised. To construct API inventory, users need to crawl standard API names from official API websites, and then extend the standard API names with commonly-seen synonyms. The identification of common synonyms is semi-automatic, based on human observation of unsupervised Brown clusters. For the API Linking of *APIReal*, users need to prepare the knowledge base for a new library. However, the knowledge base of *APIReal* is based on API inventory, which is set up in API recognition step. Therefore, it will take little effort to expand the API linking approach of *APIReal* to a new library.

Once unsupervised language models and API inventory for API recognition and knowledge base for API linking are prepared, *APIReal* can recognize API entities in a Stack Overflow post and link the recognized API mentions to corresponding API documentation effectively. To evaluate the runtime performance of *APIReal*, we randomly select 100 Stack Overflow posts with the tags of three Python libraries in our experiment and calculate the runtime cost of API recognition and linking. The average and standard deviation runtime cost per post for API recognition and linking are 7.09±5.02 and 7.38±6.24 seconds, respectively. This experiment is run on a laptop with Mac OS 10.12, 2.3 GHz Intel Core i5 CPU, and 16 GB memory. We find that the *APIReal* runtime is related to the length of a post, i.e., the longer a post, the more time *APIReal* takes. We also find that the processing step takes the most proportion of runtime cost (∽80%). Thus, for the posts that have very long content, we can split the content into several small parts, and then *APIReal* performs API recognition and linking for these small parts one by one. This could provide better user experience in practice.

8 Threats to validity

API recognition One major threat to validity of the learning-based API recognition approach of *APIReal* is human labeling of training and test sentences. The incorrect human labels would potentially have negative effects on the modeling training and testing. To alleviate this threat, the authors cross-check the labeling results and resolve any disagreements in the labeling results. However, sometimes even human cannot disambiguate whether a token is an API mention or not, especially for common nouns that refer to basic computing concepts, for example, *array* and *dataframe* which can be basic computing concepts or APIs (*Numpy*'s *array* package, *Pandas*'s *DataFrame* class). For example, *Numpy* has a *array* package. For the sentences like "I use numpy's array" or "see array package of Numpy", we can easily label the *array* as API mentions. When preparing training and test sentences, we find some sentences like "you can use array" in the post discussing the use of *Numpy*'s *array* package. Even for human, it is difficult to determine whether the user refers to the *array* package or the concept of array. Another similar case is the token dataframe, which can be the concept of tabular data structure or the *pandas.DataFrame*. In our experiments, we take a conservative strategy and do not label the token 'array' and 'dataframe' as API mentions unless both authors agree.

The API evolution is also an threat to validity we encounter in data labeling. For example, a user mentions "You can also downsample using the asof method of pandas.DateRange objects" (post ID 10020591). From the sentence context, we label *pandas.DateRange* as an

API mention. However, we cannot find *pandas.DateRange* in the official API reference of the *Pandas* library. We search the Web and find that *pandas.DateRange* is an API in an old version of *Pandas*, and has been renamed as *pandas.data_range*. In such cases, we still label the token as an API mention. However, such cases are rare.

Another threat to validity is that we only implement two simple baseline approaches to evaluate our approach *APIReal*. We find that "island parser" used in the approach of Rigby and Robillard (2013) might be a more complicated candidate baseline. However, Island parser is usually difficult to implement on code in English. The island parser usually knows the structure of the language which is fine for code snippets and uses several context filters. Most of this structure is not present when an API element is mentioned in English. Hence, We implement a baseline using regular expression and code annotation, which has been shown effectiveness on extracting API entities on Stack Overflow in several previous studies (e.g., Parnin et al. 2012 and Linares-Vásquez et al. 2014). This baseline could also be considered as a simplification of island parser because the annotated code can be considered as an "island", i.e., API mention but it does not consider the context around API mentions.

API linking The threats to validity of the API linking approach of *APIReal* in our evaluations involve: 1) the limited number of linking examples examined. We performed a preliminary evaluation to our API linking approach by manually inspecting a ground truth dataset consisting of 120 APIs for linking. However, we are able to observe similar performances across different studied libraries. In the future, we will use more APIs from different libraries to evaluate the API linking approach of *APIReal*. 2) our knowledge base only consists of 13713 records crawled from 4 libraries, which is smaller compared to that of Baker (Subramanian et al. 2014). However, Stack Overflow questions always contain at least one tag to indicate the scope of the conservation. Even if we add more API information in the knowledge base, we could always choose to simply limit the scope for linking to only a few libraries or classes using our scope filter component, as covered previously.

9 Conclusions and future work

The primary focus of this paper is to address a long-avoided challenge in API recognition, i.e., the ambiguity between the API sense and the normal sense of a common word in informal natural language sentences. We tackle the challenge by exploiting name synonyms and semantic context features derived from unsupervised word representations learned from the abundant unlabeled text. Our evaluation shows that using these features in conditional random field, together with self-training, make our approach robust and accurate for extracting common-word API mentions, even in the face of the wide presence of API-mention and sentence-context variations in informal social discussions.

Second, this paper presents an rule-based API linking approach, based on the results of the API recognition approach. We design a set of problem-specific heuristics that utilize the global context of the discussion thread of a Stack Overflow question to resolve the inherent ambiguities in API mentions. Our evaluation shows the effectiveness of the proposed linking method.

In the future, we plan to further investigate the potential of applying clone detection based methods (Abdalkareem et al. 2017), learning to link techniques (Milne and Witte 2008), as well as the recently proposed joint recognition and linking techniques (Ji et al. 2016), to the research problem of API recognition and linking in software engineering.

References

Abdalkareem R, Shihab E, Rilling J (2017) On code reuse from stackoverflow: an exploratory study on android apps. Inf Softw Technol 88:148–158

Antoniol G, Canfora G, Casazza G, De Lucia A, Merlo E (2002) Recovering traceability links between code and documentation. IEEE Trans Softw Eng (TSE) 28(10):970–983

Bacchelli A, D'Ambros M, Lanza M, Robbes R (2009) Benchmarking lightweight techniques to link e-mails and source code. In: Proceedings of the 16th working conference on reverse engineering (WCRE). IEEE, Piscataway, pp 205–214

Bacchelli A, Lanza M, Robbes R (2010) Linking e-mails and source code artifacts. In: Proceedings of the 32nd ACM/IEEE international conference on software engineering (ICSE). ACM, New York, pp 375–384

Bacchelli A, Cleve A, Lanza M, Mocci A (2011) Extracting structured data from natural language documents with island parsing. In: Proceedings of the 26th IEEE/ACM international conference on automated software engineering (ASE). IEEE, Piscataway, pp 476-479

Brown PF, Desouza PV, Mercer RL, Pietra VJD, Lai JC (1992) Class-based n-gram models of natural language. Comput Linguist 18(4):467–479

Chen F, Kim S (2015) Crowd debugging. In: Proceedings of the 10th joint meeting on foundations of software engineering (FSE). ACM, New York, pp 320–332

Chen X, Liu Z, Sun M (2014) A unified model for word sense representation and disambiguation. In: EMNLP, Citeseer, pp 1025–1035

Dagenais B, Robillard MP (2012) Recovering traceability links between an api and its learning resources. In: Proceedings of the 34th international conference on software engineering (ICSE). IEEE, Piscataway, pp 47-57

Fleiss JL (1971) Measuring nominal scale agreement among many raters. Psychol Bull 76(5):378

Gao Q, Zhang H, Wang J, Xiong Y, Zhang L, Mei H (2015) Fixing recurring crash bugs via analyzing q&a sites (t). In: Proceedings of the 30th IEEE/ACM international conference on automated software engineering (ASE). IEEE, Piscataway, pp 307–318

Guo J, Che W, Wang H, Liu T (2014) Revisiting embedding features for simple semi-supervised learning. In: EMNLP, pp 110–120

Ji Z, Sun A, Cong G, Han J (2016) Joint recognition and linking of fine-grained locations from tweets. In: Proceedings of the 25th international conference on world wide web (WWW), International World Wide Web Conferences Steering Committee, pp 1271–1281

Jiang HY, Nguyen TN, Chen X, Jaygarl H, Chang CK (2008) Incremental latent semantic indexing for automatic traceability link evolution management. In: Proceedings of the 23rd IEEE/ACM international conference on automated software engineering (ASE), IEEE Computer Society, pp 59–68

Lafferty JD, McCallum A, Pereira FCN (2001) Conditional random fields: Probabilistic models for segmenting and labeling sequence data. In: Proceedings of the Eighteenth international conference on machine learning, ICML '01, pp 282–289

Li C, Sun A (2014) Fine-grained location extraction from tweets with temporal awareness. In: Proceedings of the 37th international ACM SIGIR conference on research & development in information retrieval. ACM, New York, pp 43–52

Liang P (2005) Semi-supervised learning for natural language. PhD thesis, Citeseer

Liao W, Veeramachaneni S (2009) A simple semi-supervised algorithm for named entity recognition. In: Proceedings of the NAACL HLT 2009 Workshop on Semi-Supervised Learning for Natural Language Processing, Association for Computational Linguistics, pp 58–65

Linares-Vásquez M, Bavota G, Di Penta M, Oliveto R, Poshyvanyk D (2014) How do api changes trigger stack overflow discussions? a study on the android sdk. In: Proceedings of the 22nd international conference on program comprehension (ICPC). ACM, New York, pp 83–94

Liu X, Zhang S, Wei F, Zhou M (2011) Recognizing named entities in tweets. In: Proceedings of the 49th annual meeting of the association for computational linguistics: human language technologies-Volume 1, Association for Computational Linguistics, pp 359–367

Liu X, Li Y, Wu H, Zhou M, Wei F, Lu Y (2013) Entity linking for tweets. In: ACL (1), pp 1304–1311

Marcus A, Maletic J et al (2003) Recovering documentation-to-source-code traceability links using latent semantic indexing. In: Proceedings of the 25th international conference on software engineering (ICSE). IEEE, Piscataway, pp 125-135

Mihalcea R (2004) Co-training and self-training for word sense disambiguation. In: CoNLL, pp 33–40

Mihalcea R, Csomai A (2007) Wikify!: linking documents to encyclopedic knowledge. In: Proceedings of the sixteenth ACM conference on Conference on information and knowledge management. ACM, New York, pp 233–242

Mikolov T, Chen K, Corrado G, Dean J (2013a) Efficient estimation of word representations in vector space. arXiv preprint arXiv:13013781

Mikolov T, Sutskever I, Chen K, Corrado GS, Dean J (2013b) Distributed representations of words and phrases and their compositionality. In: Advances in neural information processing systems, pp 3111–3119

Milne D, Witte IH (2008) Learning to link with wikipedia. In: Proceedings of the 17th ACM conference on Information and knowledge management. ACM, New York, pp 509–518

Moonen L (2001) Generating robust parsers using island grammars. In: Proceedings of eighth working conference on reverse engineering (WCRE). IEEE, Piscataway, pp 13-22

Navigli R (2009) Word sense disambiguation: a survey. ACM Comput Surv (CSUR) 41(2):10

Parnin C, Treude C, Grammel L, Storey MA (2012) Crowd documentation: Exploring the coverage and the dynamics of api discussions on stack overflow. Georgia Institute of Technology, Tech Rep

Rahman MM, Roy CK, Lo D (2016) Rack: Automatic api recommendation using crowdsourced knowledge. In: SANER

Rigby PC, Robillard MP (2013) Discovering essential code elements in informal documentation. In: Proceedings of international conference on software engineering (ICSE). IEEE Press, Piscataway, pp 832–841

Shen W, Wang J, Luo P, Wang M (2012) Liege:: Link entities in web lists with knowledge base. In: Proceedings of the 18th ACM SIGKDD international conference on knowledge discovery and data mining, ACM, KDD '12, pp 1424–1432

Subramanian S, Inozemtseva L, Holmes R (2014) Live api documentation. In: Proceedings of the 36th international conference on software engineering (ICSE). ACM, New York, pp 643–652

Turian J, Ratinov L, Bengio Y (2010) Word representations: a simple and general method for semi-supervised learning. In: Proceedings of the 48th annual meeting of the association for computational linguistics, Association for Computational Linguistics, pp 384–394

Wang M, Manning CD (2013) Effect of non-linear deep architecture in sequence labeling. In: IJCNLP, pp 1285–1291

Wu D, Lee WS, Ye N, Chieu HL (2009) Domain adaptive bootstrapping for named entity recognition. In: Proceedings of the 2009 conference on empirical methods in natural language processing: Volume 3-Volume 3, Association for Computational Linguistics, pp 1523–1532

Wu N, Hou D, Liu Q (2016) Linking usage tutorials into api client code pp 22–28

Yao Y, Sun A (2015) Mobile phone name extraction from internet forums: a semi-supervised approach. World Wide Web pp 1–23

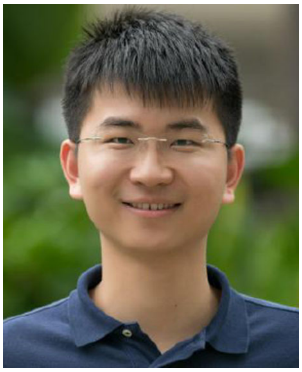
Yarowsky D (1995) Unsupervised word sense disambiguation rivaling supervised methods. In: Proceedings of the 33rd annual meeting on association for computational linguistics, association for computational linguistics, pp 189–196

Ye D, Xing Z, Foo CY, Ang ZQ, Li J, Kapre N (2016a) 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)

Ye D, Xing Z, Li J, Kapre N (2016b) Software-specific part-of-speech tagging: An experimental study on stack overflow. In: Proceedings of the 31st annual ACM symposium on applied computing, ACM, New York, SAC '16, pp 1378–1385. https://doi.org/10.1145/2851613.2851772

Yu M, Zhao T, Dong D, Tian H, Yu D (2013) Compound embedding features for semi-supervised learning. In: HLT-NAACL, pp 563–568

Zheng W, Zhang Q, Lyu M (2011) Cross-library api recommendation using web search engines. In: Proceedings of the 19th ACM SIGSOFT symposium and the 13th European conference on foundations of software engineering. ACM, New York, pp 480–483



Deheng Ye Ph.D. is currently a Senior Researcher in Tencent AI Lab, Shenzhen, China. He obtained his PhD degree from the School of Computer Science and Engineering, Nanyang Technological University, Singapore, in 2017. His research interests lie in data mining in software engineering, including software-specific text mining and source code mining.

Lingfeng Bao is currently a postdoctoral research fellow in the College of Computer Science and Technology, Zhejiang University. He received his B.E. and PhD degrees both from the College of Software Engineering, Zhejiang University, in 2010 and 2016, respectively. His research interests are software analytics, behavioral research methods, data mining, and human computer interaction.

Dr. Zhenchang Xing is now a Senior Lecturer in the Research School of Computer Science, Australian National University. Previously, he was an Assistant Professor in the School of Computer Science and Engineering, Nanyang Technological University, Singapore, from 2012-2016. Dr. Xing's research interests include software engineering, data mining and human-computer interaction. His work combines software analytics, behavioral research methods, data mining techniques, and interaction design to understand how developers work, and then build recommendation or exploratory search systems for the timely or serendipitous discovery of the needed information.

Shang-Wei Lin Ph.D. received his B.S. degree in Information Management from the National Chung Cheng University, Chiayi, Taiwan, in 2003 and received his Ph.D. degree in Computer Science and Information Engineering from the National Chung Cheng University, Chiayi, Taiwan, in 2010. From September 2003 to July 2010, he was a teaching and research assistant in the Department of Computer Science and Information Engineering at the National Chung Cheng University. In 2011, he was a postdoctoral researcher at School of Computing, National University of Singapore (NUS). From 2012 to November 2014, he was a research scientist at Temasek Laboratories in National University of Singapore (NUS). From December 2014 to April 2015, he was a postdoctoral research fellow in Singapore University of Technology and Design (SUTD). He has joined School of Computer Science and Engineering, Nanyang Technological University (NTU) as Assistant Professor in April 2015. His research interests include formal verification, formal synthesis, embedded system design, cyberphysical systems, security systems, multi-core programming, and component-based object-oriented application frameworks for real-time embedded systems. Recently, he is also working on developing formal verification techniques to validate Ethereum smart contracts.