Learning to Extract API Mentions from Informal Natural Language Discussions

Deheng Ye, Zhenchang Xing, Chee Yong Foo, Jing Li, and Nachiket Kapre
School of Computer Science and Engineering
Nanyang Technological University, Singapore
ye0014ng@e.ntu.edu.sg, {zcxing, fooc0029, jli030, nachiket}@ntu.edu.sg

***Abstract*—When discussing programming issues on social platforms (e.g, Stack Overflow, Twitter), developers often mention APIs in natural language texts. Extracting API mentions in natural language texts is a prerequisite for effective indexing and searching for API-related information in software engineering social content. However, the informal nature of social discussions creates two fundamental challenges for API extraction: *common-word polysemy* and *sentence-format variations*. Common-word polysemy refers to the ambiguity between the API sense of a common word and the normal sense of the word (e.g., append, apply and merge). Sentence-format variations refer to the lack of consistent sentence writing format for inferring API mentions. Existing API extraction techniques fall short to address these two challenges, because they assume distinct API naming conventions (e.g., camel case, underscore) or structured sentence format (e.g., code-like phrase, API annotation, or full API name). In this paper, we propose a semi-supervised machine-learning approach that exploits name synonyms and rich semantic context of API mentions to extract API mentions in informal social text. The key innovation of our approach is to exploit two complementary unsupervised language models learned from the abundant unlabeled text to model sentence-format variations and to train a robust model with a small set of labeled data and an iterative self-training process. The evaluation of 1,205 API mentions of the three libraries (*Pandas, Numpy, and Matplotlib*) in Stack Overflow texts shows that our approach significantly outperforms existing API extraction techniques based on language-convention and sentence-format heuristics and our earlier machine-learning based method for named-entity recognition.**

I. Introduction

APIs are an important resource for software engineering. APIs appear not only in code, but also in natural language texts, such as formal API specifications and tutorials, as well as developers' informal discussions, such as emails and online Q&A posts. In this paper, we are concerned with extracting API mentions from informal natural language texts such as Stack Overflow discussions. For example, Fig. 1 shows a sentence from a Stack Overflow post (post ID is 12182744). In this sentence, we would like to extract the *series* and the second *apply* as API mentions, which are a class name and a method name of the Pandas Library, respectively. Extracting such fine-grained API mentions is a prerequisite for indexing, analyzing, and searching informal natural language discussions for software engineering tasks, such as API linking [1], [2], [3], [4], API recommendation [5], and bug fixing [6], [7].

Indeed, the importance of fine-grained API extraction from natural language sentences has long been recognized. Representative techniques include language-convention based regular expressions [8], [1], [2] and island parsing [9], [3]. 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 [2]. However, they fall short to address the following two fundamental challenges in API extraction from *informal* natural language texts:

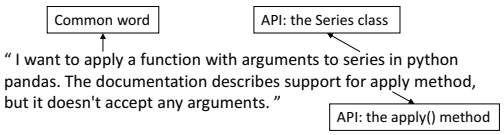

Fig. 1: Illustrating Our Task

- *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 name, such as the *Series* class and the *apply* method. When such APIs are mentioned with distinct orthographic features, such as *pandas.Series*, *apply()*, and *<code>apply</code>*, 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 [10], [11]) 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 API mentions (labeling confidence score > 0.8). This creates the challenge in *disambiguating 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. For API extraction from developers' informal discussions, e.g., emails, Bacchelli et al. [1] have shown that this challenge poses a big threat to language-convention based regular expressions, as no observational orthographic features can be utilized. However, this common-word polysemy challenge is generally avoided by considering only APIs mentions with distinct orthographic features in existing

978-1-5090-3806-0/16 $31.00 © 2016 IEEE
DOI 10.1109/ICSME.2016.11

TABLE I: A Subset of Variant Forms of API Mentions

Writing Form	Freq.	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 II: 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.

work [3]. There has been no work from the software engineering community addressing the polysemy problem in fine-grained API extraction.

- *Sentence-format variations:* Informal discussions on social platforms (such as Stack Overflow, Twitter) 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 I 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 II, there lacks of consistent use 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 from non-API words. However, as shown in Table II, 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 [12], [13] in informal text (i.e., variations that have not been seen in the training data even when a huge amount of data has been examined).

In this paper, we propose a semi-supervised machine learning approach to solve the problem. To model API-mention and sentence-context variations, our approach exploits state-of-the-art unsupervised language models, in particular class-based Brown Clustering [14], [15] and neural-network-based word embedding [16], [17] 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 Random Field (CRF) model [18], together with an iterative self-training mechanism (a.k.a. bootstrapping) [19], our approach requires only a small set of human labeled sentences to train a robust model for extracting API mentions in informal natural language sentences.

To evaluate our approach, we choose to extract APIs for three Python libraries, i.e., Pandas, Numpy and Matplotlib, because these libraries 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 compare our approach with three state-of-the-art methods for API extraction from natural language text, including lightweight regular expressions [1], island parsing [3], and machine-learning based software-specific entity recognition [11]. Our approach consistently and significantly outperforms the three baseline methods.

II. Related Work

Many software engineering tasks require or benefit from fine-grained API extraction techniques, such as API linking (a.k.a traceability recovery) [20], [21], [22], [8], [1], [2], [3], [4], API recommendation [5], [23] and bug fixing [6], [7]. In this section, we discuss the state-of-the-art methods for fine-grained API extraction from natural language texts.

Bacchelli et al. [1], [8] develop an API extraction and linking infrastructure, called Miler. They use lightweight regular expressions of distinct orthographic features and information retrieval techniques to detect class and method mentions in developer emails. They show that information retrieval techniques do not work for fine-grained API extraction task, whose performance is even significantly worse than lightweight regular expressions. Furthermore, their study shows that common-word polysemy and non-standard API synonyms significantly degrade the performance of lightweight regular expressions.

Dagenais and Robillard [2] develop RecoDoc to extract Java APIs from several learning resources (formal API documentation, tutorial, forum posts, code snippets) and then perform traceability link recovery across different sources. They devise

a pipeline of filters to resolve the traceability link ambiguities. However, they extract API mentions from natural language text using regular expressions similar to those of Miler [1]. That is, their API extraction from natural language text again relies 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) [24]. Bacchelli et al. [9] extract coarse-grained structured code fragments from natural language text with island parsing. Rigby and Robillard [3] also use island parser to identify code-like elements that can potentially be APIs. They further resolve the code-like phrases to fine-grained APIs. However, simple names of methods that are not suffixed by () are simply ignored [3]. For example, they consider only *HttpClient.execute* and *execute()* as API mentions, but ignore the single word *execute* which also likely refers to the same API.

Our work is related to two lines of work in natural language processing, named entity recognition (NER) whose goal is to extract and categorize entities (e.g., location, people) in natural language text [13], [12], [25], and word sense disambiguation (WSD) whose goal is to disambiguate the sense of polysemous words in a given sentence context [26], [27], [28]. Recently, Ye et al. [11] propose 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. Using S-NER for the task of API extraction has several limitations: 1) it aims to recognize a broad category of software entities, making it 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.

III. Approach Overview

In this section, we formulate our research problem of API extraction from informal natural language text, and give an overview of our proposed approach.

A. Problem Formulation

Given a natural language sentence (e.g., from Stack Overflow posts), our task is to recognize all API mentions in the sentence, as illustrated in the example in Fig. 1. Specially, 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 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, for example, *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 the *Pandas* library, Figure (class) and draw (method) of *Matplotlib*, and Polynomial (class) and flatten (method) of *Numpy*.

Note that in this work we focus on tackling common-word polysemy and sentence-format variations issues in recognizing whether a token is an API mention of certain libraries. A related research problem is to link the recognized API mentions to the corresponding API of a specific library, which is referred to as API linking [2], [3], [4]. The task of API linking is beyond the scope of this paper.

B. Overview of Main Steps

Fig. 2 shows the main steps of our approach. In this study, we obtain natural language sentences from Stack Overflow discussions[1] to train and evaluate our approach (Section IV-A).

In our approach, we use two unsupervised language models (i.e., class-based Brown clustering [14] and neural-network based word embedding [17], [16]) to learn word representations from unlabeled text and cluster semantically similar words (Section IV-D). We exploit word representations and clusters in three perspectives. First, we observe word clusters that contain standard API names to infer commonly used library or class name synonyms, for example, *pd* for *pandas*, *df* for *DataFrame*. A collection of standard API names and the inferred common API synonyms constitute an API inventory for a specific library (Section IV-B). Second, unsupervised word representations are used to represent semantically equivalent API-mention variations and sentence-context variations, which help address the out-of-vocabulary issue (Section IV-D). Third, unsupervised word clusters, together with self-training process (Section IV-E), help alleviate the lack of labeled data for modeling training.

Our approach trains a linear-chain Conditional Random Field (CRF) model using orthographic features from tokens, compound word-representation features from the two different unsupervised language models, and gazetteer feature from the API inventory (Section IV-F). The training starts with a small set of human labeled sentences to obtain an initial CRF model. Then, the training continues through an iterative self-training process over a large set of unlabeled sentences (Section IV-E), through which high-confidence machine labeled sentences, together with human labeled sentences, are used to retrain the CRF model.

[1]All the data used is from Stack Overflow January 2016 data dump.

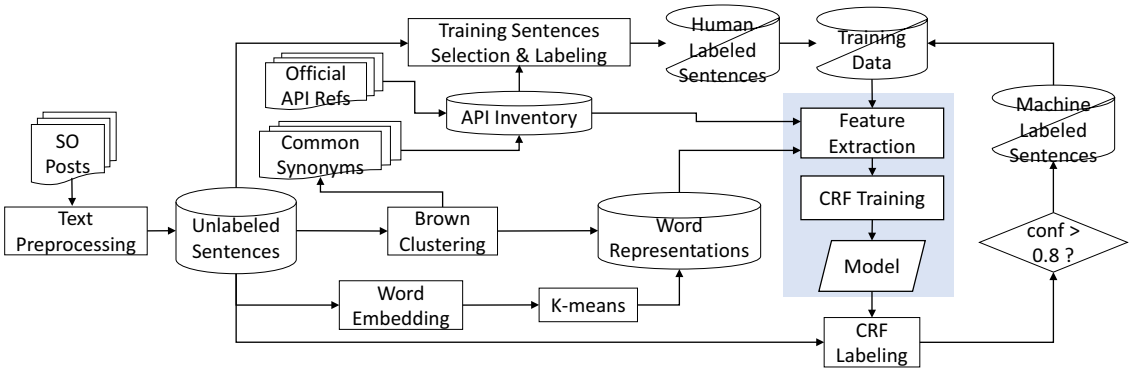

Fig. 2: The Overview of Our Approach

IV. Approach Details

We now describe each component of our approach in detail.

A. Text Preprocessing

Given a Stack Overflow post, the preprocessing steps include code snippet removal, HTML tags cleaning, and tokenization, which is similar to [29]. We remove large code snippets in <pre><code>, but not short code elements in <code> in natural language sentences. We write a sentence parser to split the post text into sentences. We use our software-specific tokenzier [10] to tokenize the sentences. This tokenizer preserves the integrity of code-like tokens. For example, it treats *pandas.DataFrame.apply()* as a single token, instead of a sequence of 7 tokens, i.e., *pandas . DataFrame . apply ( )*. For other text (e.g., email), different preprocessing steps, sentence parser, and tokenizer may be needed.

B. Constructing API Inventory

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 [11] shows that a gazetteer of standard API names contributes only marginally to the NER performance. This is partially because of the wide presence of non-standard API synonyms and their derivational forms in informal natural language text (see Table I for examples). Therefore, we construct an API inventory for a library that contains not only standard API names but also their commonly-seen synonyms.

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 [13], [12] and WSD [27], [26] 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 (see Section IV-D) 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 (see Section IV-F); 2) ensure that training data and test data reaches a good coverage of polysemous and derivational forms of library APIs (see Section IV-C).

C. Training Sentences Selection and Labeling

The quality and amount of human labeled data for model training are essential to the performance of a machine learning system. However, there has 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 disambiguating 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 [28], [26].

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 [12] and the mobile phone name extraction work [30], 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 be API mentions but can 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 Section IV-E.

D. Learning Word Representations

In informal social discussions, both API mentions and sentence context vary greatly (see Table I and Table II). These variations result in out-of-vocabulary issue [12] 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.

The unsupervised language models used in this paper include class-based Brown clustering [14], [15] and neural-network based word embedding [16]. Different language models make different assumptions about corpus properties to evaluate the semantic similarity between words. Empirical studies [31] and [32] show that Brown clustering and word embedding produce complementary views 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.

Assume 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 as described in Section IV-A. This produces a large set of unlabeled sentences. Unlike prior work [31], [32], [11], [12], [30], 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 [14] outputs a collection of word clusters. Each word belongs to one cluster. Words in the same cluster share the same bitstring representation. Studies [30], [11], [12] show that Brown clusters are useful for identifying abbreviations and synonyms. Indeed, we exploit Brown clusters learned from unlabeled text to expand standard API names with commonly-seen name synonyms (see Section IV-B).

For neural-network based word embedding, we use continuous skip-gram model [16], [33] 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 [16], [17]. However, studies [31], [34], [32] 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 [31], [32], 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 [31], 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 dimension is set to 0.

Word representations obtained from the Brown clusters and the word embedding clusters are used as features to the CRF model (see Section IV-F). This helps the CRF model tolerate semantically similar API-mention and sentence-context varia-

tions, and thus alleviate the out-of-vocabulary issue.

E. Iterative Self-Training Process

The quality of a machine learning model relies on the sufficient, high-quality training data. In this work, we only manually label a small set of training data (see Section IV-C). 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 [19], 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.

Algorithm 1 outlines the self-training process. The algorithm first trains a CRF classifier using the small set of human labeled data obtained in Section IV-C (lines 1-2). The details of the CRF classifier are explained in Section IV-F. 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 executed.

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 ← human labeled sentences;
2  l = train(te);
3  for S ∈ unlabeledsents && iterations < M do
4      S⃗ = feature_extractor(S);
5      (S_labeled, conf) = crf_label(l, S⃗);
6      if conf > α then
7          te ← te ∪ {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
```

F. CRF-based Classifier

Given a token in a natural language sentence, our approach determines whether the token is an API mention or a normal word using a linear-chain Conditional Random Fields (CRF) [18]. The CRF classifier is the state-of-the-art model for sequential labeling, which is particularly strong at learning from 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.

In this work, we design three kinds of features for the CRF classifier: *orthographic features* of current tokens and its surrounding tokens, *word-representation features* of current token (word) 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_iw_{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 to a cluster ID. We denote the cluster ID of the current word as c_i. Following the pioneer work of utilizing compound cluster features [31], [32], 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 of the entries ends with w_nb; 3) if w_nb contains dot in the middle, we consider it as a match if any of the entries begins 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.

V. Experiment Setup

This section describes the tools we use to implement our approach, studied libraries, model training settings, testing dataset labeling, evaluation metrics, and the baseline methods we compare with.

A. Tool Implementation

We implement web crawlers using Scrapy [35] to crawl official API names. For the implementation of the linear CRF, we use CRFsuite [36], a popular CRF toolkit for sequential labeling. For Brown Clustering, we use Liang's implementation [37]. We learn continuous word embeddings using word2vec [38], which contains an efficient open-source implementation of the skip-gram model [16]. We use the K-means implementation from Sofia-ML [39] to perform K-means clustering of the continuous word embeddings.

B. Studied libraries

The key challenge we aim to address is to disambiguate the API sense of a common word and the normal sense of the word in a natural language sentence. To evaluate whether our approach 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 III 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 as described in Section IV-B. APIs in the inventory are then split into a non-polysemous set and a polysemous set (see Section IV-C). *Pandas*, *Numpy* and *Matplotlib* have 55.04%, 16.04% and 41.36% APIs whose simple name is polysemous common word, respectively.

Another important reason we choose these three libraries is that the APIs of the three libraries are for diverse functionalities: *Pandas* is for panel data manipulation and analysis, *Numpy* for scientific computing, *Matplotlib* for 2D plotting. Semantically, *Matplotlib* is more distant from *Pandas* than *Numpy*. Using these three libraries helps demonstrate the generality of our approach.

C. Model Training

Human labeled sentences for model training are from `Pandas` posts only. The sentence selection and labeling process is described in Section IV-C. The labeling results are cross-checked by the first and third author to reach agreements on the labels. Unlabeled sentences used to learn Brown clusters and word embeddings include all the sentences from Stack Overflow posts that are tagged with `pandas`, `numpy` and `matplotlib`, except those sentences selected as training and test sentences. 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 [31], 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 (Section IV-F).

For the self-training process, we randomly select unlabeled sentences from the `pandas` discussions 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 [26], [25], [13]. 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.

D. Human Labeling of Testing Data

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

At 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 IV 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.

It is important to note that our training sentences contain only mentions of *Pandas*'s APIs, while our test sentences contain mentions of APIs of not only *Pandas* but also *Numpy* and *Matplotlib*. To avoid model overfitting, out test data does not contain Pandas's APIs mentioned in the training data. Furthermore, the mentions of *Numpy*'s and *Matplotlib*'s APIs are completely new to the CRF model trained using mentions of *Pandas*'s APIs.

E. Evaluation metrics

We use precision, recall, and F1-score to evaluate the performance of an API extraction method. Precision measures

TABLE III: Information of the Studied Libraries

Library	Version	#SO Questions	Attribute	#APIs	#Polysemous API	Percentage
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	1.11.0	24,390	Scientific computing	2217	917	41.36%

TABLE IV: 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

what percentage the recognized APIs by a method are correct. Recall measures what percentage the API mentions in the testing dataset are recognized correctly by a method. F1-score is the harmonic mean of precision and recall.

F. Baseline Methods for API Extraction

We compare our approach with three state-of-the-art methods for fine-grained API extraction from natural language text.

- **Baseline1 - Lightweight regular expressions**. We implement the lightweight regular expressions used in Miler [1]. 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 as 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 [1] for details).
- **Baseline2 - Island parsing**. We implement island parsing following the descriptions in Section 4 of [3], except that their parser is based on Java language specification, while ours is based on Python language convention. Note that in [3] authors examine the validity of the extracted APIs using API linking techniques from RecoDoc [2], while such API linking techniques are not used in this baseline, as API linking is out of the scope of this paper. Our island parser also exploits code annotation practice on Stack Overflow, i.e., annotate small code fragment in text using HTML tag <code>. Our island parsing considers a single token that is annotated with <code> as an "island", i.e., an API mention.
- **Baseline3 - Machine-learning based NER**. We use the software-specific entity recognition tool (S-NER) proposed in our earlier work [11] to recognize the API mentions in our testing data. For fair comparison, we use the same set of features used in [11], and re-train the model of S-NER with the same set of human labeled sentences for training the CRF model of this work.

VI. Experiment Results and Analysis

We now report experiment results and analyze our findings.

A. Overall Results for All API Mentions

Table V shows the comparison of the three evaluation metrics for using the three baseline methods and our method to extract all API mentions in the testing dataset. Our method outperforms all the baseline methods. It achieves the best and balanced precision and recall, and the F1-score is 0.876.

TABLE V: Comparison of Overall Performance

Method	Precision	Recall	F1-score
Baseline1	0.125	0.723	0.213
Baseline2	0.633	0.624	0.628
Baseline3	0.825	0.678	0.744
Our method	0.879	0.872	0.876

Comparison with Baseline1: we observe almost the same performance result as that of Miler [1] 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 Baseline1 (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 Baseline1 methods cannot properly address common-word polysemy issue.

Comparison with Baseline2: The Baseline2, i.e., a carefully designed island parser that exploits language conventions and sentence structures (e.g., code annotation), proves to be more useful and reliable for API extraction from informal text. The island parsing baseline achieves balanced precision and recall, and the F1-score is 3 times higher than that of the Baseline1. However, it still misses about 38% of the API mentions and about 37% of the extracted API mentions are not true API mentions. Island parsing 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.

Comparison with Baseline3: The Baseline3, i.e., machine-learning based software-specific named entity recognition,

TABLE VI: API Extraction Performance for Each of the 3 Studied Libraries

Method	Pandas			Matplotlib			Numpy		
	Precision	Recall	F1-score	Precision	Recall	F1-score	Precision	Recall	F1-score
Baseline1	0.153	0.791	**0.257**	0.107	0.689	**0.180**	0.111	0.675	**0.191**
Baseline2	0.640	0.615	**0.627**	0.611	0.622	**0.617**	0.617	0.645	**0.631**
Baseline3	0.858	0.791	**0.823**	0.779	0.527	**0.629**	0.795	0.705	**0.747**
Our Method	0.913	0.889	**0.901**	0.856	0.879	**0.867**	0.847	0.873	**0.860**

achieves significantly higher precision and a moderate improvement on recall, compared with the Baseline2. Our method can improve the precision even further, and meanwhile significantly improve the recall. Our method's improvement on recall over Baseline3 can be attributed to the use of unsupervised word representations as compound semantic context features. In contrast, the Baseline3 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 Baseline3's improvement on recall over the island parsing is moderate. Other new features introduced in our method, such as commonly-seen synonyms in API inventory, self-training, and two complementary word representations, also contribute to boosting up precision and recall, compared with our previous machine-learning based method [11].

B. Results for Each of the Studied Libraries

Table VI 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 Baseline3 and our method (i.e., two different machine-learning based methods) *across* libraries. The Baseline3 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 749 and 0.629 respectively). Similarly, our method also performs the best for *Pandas*'s APIs (F1-score 0.901), but the performance of our method drops only slightly for *Numpy*'s and *Matplotlib*'s APIs (F1-score 0.860 and 0.867 respectively).

Recall that we train the Baseline3's model 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. Therefore, the performance of the Baseline3 drops significantly, especially for *Matplotlib* which is more distant from *Pandas* than *Numpy*.

TABLE VII: 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 and gazetteer features	0.745	0.447	0.559

In addition to orthographic features and semantic representations of API mentions, our method 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 our method more robust than the Baseline3 for extracting mentions of *Numpy*'s and *Matplotlib*'s APIs.

C. Feature Ablation

We ablate one kind of feature(s) at a time from our full feature set (see Section IV-F) and study the impact of different kinds of features on the API extraction performance. Table VII 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.

This result implies that the performance of our method 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, our approach still outperforms the best baseline method (i.e., Baseline3). However, when ablating both features from word representations and API inventory, i.e., only orthographic features are retained, the performance of our method deteriorates significantly, and becomes worse than both the Baseline3 and the Baseline2. This indicates the importance of our word-representation and gazetteer features.

D. The Impact of Self-Training

When training the CRF model, we perform 10 iterations of self-training. Figure 3 shows the change of the F1-score for extracting all API mentions after each iteration. We can

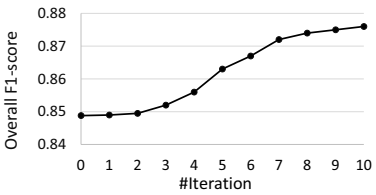

Fig. 3: The Impact of Self-training

see that even without self-training, the CRF model trained using only human labeled sentences (i.e., at #iteration=0) still outperforms the Baseline3. With the increase of self-training iterations, the F1-score increases monotonically. From the iteration 1 to 3 and from the iteration 8 to 10, the F1-score is relatively stable (with less than 0.2% increase). However, the F1-score increases about 2% from the iteration 4 to 7. This implies that the retraining of the model with machine labeled sentences in the first few iterations does not expand the model much. As sufficient machine labeled sentences are accumulated after the first few iterations, the self-training starts taking effect. However, the effect diminishes after several iterations, and then the model becomes relatively stable.

Self-training works as it exposes the model with similar but not exactly the same sentences as those already in the training set (recall that we set α at 0.8, not 1.0). If the newly added sentences are almost the same as the existing sentences in the training set, it reinforces the model. If the newly added sentences have small differences from the existing ones, these small differences will be captured by the model once enough instances have been accumulated. In this way, the model is expanded with new knowledge that has not been seen in the existing training set. However, there is an upper bound of self-training, after which retraining the model with more machine labeled sentences makes little impact on the performance.

VII. Discussion

Finally, we discuss the generality of our approach and the threats to validity of our study.

A. The Generality of Our Approach

Our experiments demonstrate the generality of our approach for extracting API mentions of three very different Python libraries from Stack Overflow sentences. To expand our approach 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.

Feature ablation experiments show that with unsupervised language models, our approach can already achieve good performance. With small effort to construct the API inventory, the performance can be further boosted up. 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. To better support cross-library (or even cross-language) API extraction, we leave it as a future work to explore domain-adaptive self-training [19]. Our current self-training mechanism is a simple process, while domain-adaptive self-training aims to capture the entity differences and sentence context variances when transferring from one domain to another.

B. Threats to Validity

A major threat to validity of our approach 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-checked the labeling results and resolved any disagreements in the labeling results. However, sometimes even humans 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). In our experiments, we take a conservative strategy and do not label the token 'array' and 'dataframe' as API mentions unless both authors agree.

Another issue we encounter in data labeling is the API evolution. For example, a user mentions "You can also downsample using the asof method of pandas.DateRange objects". From the sentence context, we label *pandas.DateRange* as an API mention. However, we could not find *pandas.DateRange* in the official API reference of the *Pandas* library. We searched the Web and found 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 labeled the token as an API mention. However, such cases are rare.

VIII. Conclusion

This paper addresses a long-avoided challenge in API extraction, 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 as features in the conditional random field model, together with self-training, makes our approach robust and accurate for extracting fine-grained common-word API mentions, even in the face of the wide presence of API-mention and sentence-context variations in informal social discussions. In the future, we will investigate downstream applications that could be enabled by our fine-grained API extraction technique, including API linking, API search, and API-related issue-solution mining in software engineering social content.

Acknowledgment. This work was partially supported by Singapore MOE AcRF Tier-1 grant M4011165.020.

REFERENCES

[1] A. Bacchelli, M. Lanza, and R. Robbes, "Linking e-mails and source code artifacts," in *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering-Volume 1*. ACM, 2010, pp. 375–384.

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

[3] P. C. Rigby and M. P. Robillard, "Discovering essential code elements in informal documentation," in *Proceedings of the 2013 International Conference on Software Engineering*. IEEE Press, 2013, pp. 832–841.

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

[5] M. M. Rahman, C. K. Roy, and D. Lo, "Rack: Automatic api recommendation using crowdsourced knowledge," in *SANER*, 2016.

[6] F. Chen and S. Kim, "Crowd debugging," in *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*. ACM, 2015, pp. 320–332.

[7] Q. Gao, H. Zhang, J. Wang, Y. Xiong, L. Zhang, and H. Mei, "Fixing recurring crash bugs via analyzing q&a sites (t)," in *Automated Software Engineering (ASE), 2015 30th IEEE/ACM International Conference on*. IEEE, 2015, pp. 307–318.

[8] A. Bacchelli, M. D'Ambros, M. Lanza, and R. Robbes, "Benchmarking lightweight techniques to link e-mails and source code," in *Reverse Engineering, 2009. WCRE'09. 16th Working Conference on*. IEEE, 2009, pp. 205–214.

[9] A. Bacchelli, A. Cleve, M. Lanza, and A. Mocci, "Extracting structured data from natural language documents with island parsing," in *Automated Software Engineering (ASE), 2011 26th IEEE/ACM International Conference on*. IEEE, 2011, pp. 476–479.

[10] D. Ye, Z. Xing, J. Li, and N. Kapre, "Software-specific part-of-speech tagging: An experimental study on stack overflow," in *Proceedings of the 31st Annual ACM Symposium on Applied Computing*, ser. SAC '16. New York, NY, USA: ACM, 2016, pp. 1378–1385.

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

[12] C. Li and A. Sun, "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, 2014, pp. 43–52.

[13] X. Liu, S. Zhang, F. Wei, and M. Zhou, "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, 2011, pp. 359–367.

[14] P. F. Brown, P. V. Desouza, R. L. Mercer, V. J. D. Pietra, and J. C. Lai, "Class-based n-gram models of natural language," *Computational linguistics*, vol. 18, no. 4, pp. 467–479, 1992.

[15] P. Liang, "Semi-supervised learning for natural language," Ph.D. dissertation, Citeseer, 2005.

[16] T. Mikolov, I. Sutskever, K. Chen, G. S. Corrado, and J. Dean, "Distributed representations of words and phrases and their compositionality," in *Advances in neural information processing systems*, 2013, pp. 3111–3119.

[17] J. Turian, L. Ratinov, and Y. Bengio, "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, 2010, pp. 384–394.

[18] J. D. Lafferty, A. McCallum, and F. C. N. Pereira, "Conditional random fields: Probabilistic models for segmenting and labeling sequence data," in *Proceedings of the Eighteenth International Conference on Machine Learning*, ser. ICML '01, 2001, pp. 282–289.

[19] D. Wu, W. S. Lee, N. Ye, and H. L. Chieu, "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, 2009, pp. 1523–1532.

[20] G. Antoniol, G. Canfora, G. Casazza, A. De Lucia, and E. Merlo, "Recovering traceability links between code and documentation," *Software Engineering, IEEE Transactions on*, vol. 28, no. 10, pp. 970–983, 2002.

[21] A. Marcus, J. Maletic *et al.*, "Recovering documentation-to-source-code traceability links using latent semantic indexing," in *Software Engineering, 2003. Proceedings. 25th International Conference on*. IEEE, 2003, pp. 125–135.

[22] H.-Y. Jiang, T. N. Nguyen, X. Chen, H. Jaygarl, and C. K. Chang, "Incremental latent semantic indexing for automatic traceability link evolution management," in *Proceedings of the 2008 23rd IEEE/ACM International Conference on Automated Software Engineering*. IEEE Computer Society, 2008, pp. 59–68.

[23] W. Zheng, Q. Zhang, and M. Lyu, "Cross-library api recommendation using web search engines," in *Proceedings of the 19th ACM SIGSOFT symposium and the 13th European conference on Foundations of software engineering*. ACM, 2011, pp. 480–483.

[24] L. Moonen, "Generating robust parsers using island grammars," in *Reverse Engineering, 2001. Proceedings. Eighth Working Conference on*. IEEE, 2001, pp. 13–22.

[25] W. Liao and S. Veeramachaneni, "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, 2009, pp. 58–65.

[26] R. Mihalcea, "Co-training and self-training for word sense disambiguation." in *CoNLL*, 2004, pp. 33–40.

[27] R. Navigli, "Word sense disambiguation: A survey," *ACM Computing Surveys (CSUR)*, vol. 41, no. 2, p. 10, 2009.

[28] X. Chen, Z. Liu, and M. Sun, "A unified model for word sense representation and disambiguation." in *EMNLP*. Citeseer, 2014, pp. 1025–1035.

[29] D. Ye, Z. Xing, and N. Kapre, "The structure and dynamics of knowledge network in domain-specific q&a sites: a case study of stack overflow," *Empirical Software Engineering*, pp. 1–32, 2016.

[30] Y. Yao and A. Sun, "Mobile phone name extraction from internet forums: a semi-supervised approach," *World Wide Web*, pp. 1–23, 2015.

[31] J. Guo, W. Che, H. Wang, and T. Liu, "Revisiting embedding features for simple semi-supervised learning." in *EMNLP*, 2014, pp. 110–120.

[32] M. Yu, T. Zhao, D. Dong, H. Tian, and D. Yu, "Compound embedding features for semi-supervised learning." in *HLT-NAACL*, 2013, pp. 563–568.

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

[34] M. Wang and C. D. Manning, "Effect of non-linear deep architecture in sequence labeling." in *IJCNLP*, 2013, pp. 1285–1291.

[35] "Scrapy, http://scrapy.org/."

[36] "Crfsuite, http://www.chokkan.org/software/crfsuite/."

[37] "Brown clustering, https://github.com/percyliang/brown-cluster."

[38] "word2vec, https://code.google.com/archive/p/word2vec/."

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