2016 IEEE 23rd International Conference on Software Analysis, Evolution, and Reengineering

Software-specific Named Entity Recognition in Software Engineering Social Content

Deheng Ye, Zhenchang Xing, Chee Yong Foo, Zi Qun Ang, Jing Li, and Nachiket Kapre
School of Computer Engineering
Nanyang Technological University, Singapore
Email: ye0014ng@e.ntu.edu.sg, {zcxing, fooc0029, zang004, jli030, nachiket}@ntu.edu.sg

***Abstract*—Software engineering social content, such as Q&A discussions on Stack Overflow, has become a wealth of information on software engineering. This textual content is centered around software-specific entities, and their usage patterns, issues-solutions, and alternatives. However, existing approaches to analyzing software engineering texts treat software-specific entities in the same way as other content, and thus cannot support the recent advance of entity-centric applications, such as direct answers and knowledge graph. The first step towards enabling these entity-centric applications for software engineering is to recognize and classify software-specific entities, which is referred to as Named Entity Recognition (NER) in the literature. Existing NER methods are designed for recognizing person, location and organization in formal and social texts, which are not applicable to NER in software engineering. Existing information extraction methods for software engineering are limited to API identification and linking of a particular programming language. In this paper, we formulate the research problem of NER in software engineering. We identify the challenges in designing a software-specific NER system and propose a machine learning based approach applied on software engineering social content. Our NER system, called S-NER, is general for software engineering in that it can recognize a broad category of software entities for a wide range of popular programming languages, platform, and library. We conduct systematic experiments to evaluate our machine learning based S-NER against a well-designed rule-based baseline system, and to study the effectiveness of widely-adopted NER techniques and features in the face of the unique characteristics of software engineering social content.**

I. Introduction

Social online communities, such as Stack Overflow and Quora, play a significant role in knowledge sharing and acquisition for software developers [1]. The user-generated content in these websites has grown into an important information resource on the Web that complements traditional technical documentations [2]. A fundamental task for reusing content in these websites is searching for discussions of a specific software entity (e.g., a library, a tool, an API), to find good usage patterns, bug solutions, or alternatives. Existing approaches treat software engineering social content as textual documents, and use vector space model (e.g., TF-IDF), topic model (e.g., LDA [3]), or neural network language model (e.g., word embedding [4]) to index the content.

Existing approaches have an important limitation: *uniform importance assumption*. That is, mentions of software-specific entities in the content are treated in the same way as other regular textual content. This assumption may result in less desirable indexing of the content, because traditional information retrieval concepts such as term frequency do not apply naively to recognize essential domain-specific entities [5], [6], [4].

The social-technical nature of software engineering social content available on social information sharing websites calls for innovative forms of information extraction, organization, and search. A very desirable goal would be to organize the information in a knowledge base about different software-specific entities and relationships between entities. Such knowledge base can be represented as a graph, also known as *knowledge graph* [7]. Search systems can exploit knowledge graph not only for finding the content that actually discusses a particular software-specific entity, but also to displaying additional facts and direct information about the central entity in a query [8]. As the first step towards knowledge graphs and entity-centric search systems for software engineering domain, we must be able to recognize mentions of software-specific entities in software engineering social content and classify them into pre-defined categories. This task is referred to as *named entity recognition* or *NER* for short.

NER has been extensively studied on formal text (such as news articles [9]), informal text (such as emails [10], [11]), and social content (such as Tweets [12], [13]). The goal is to recognize real-world objects in texts, such as person, location, and organization. Some NER work from other domain exists for recognizing domain-specific entities, such as Biomedical NER [14], [15], NER in clinical notes [16]. In contrast, our study focuses on designing domain-specific NER methods for software engineering social content, a new genre of social-technical texts. Proposed solutions to NER fall into three categories: rule-based, machine learning based, and hybrid methods. Existing studies show that machine learning based methods usually outperform rule-based methods [9], [12], [16]. However, for software engineering texts, existing approaches are limited to dictionary look-up and rule-based methods based on code or text parsing techniques [17], [18], [19], [20]. Furthermore, the entity category is limited to only API.

In this work, we aim to design and evaluate a machine learning based method for general NER in software engineering social content. By general NER for software engineering, we mean that we would like to recognize not only APIs but also other categories of software-specific entities (such as programming languages, platforms, tools, libraries, frameworks, software standards). We have the following research questions:

- What are the challenges in NER for software engineering

978-1-5090-1855-0/16 $31.00 © 2016 IEEE
DOI 10.1109/SANER.2016.10

social content, compared with formal text, other social content like Tweets, or other domain-specific texts like clinical notes?
- How can we adapt state-of-the-art machine learning based NER pipeline for NER in software engineering social content?
- How well will the machine learning based NER method work in face of the unique challenges of software engineering social content?

To answer these research questions, we make the following contributions:

- We perform a formative study of a diverse set of Stack Overflow posts covering major programming languages, platforms, and libraries. Through this formative study, we identify design challenges and key design decisions to be made in NER for software engineering social content.
- We develop a software-specific, machine-learning based NER method, called S-NER, including software-specific entity categories, a software-specific tokenizer, Conditional Random Fields (CRF) based learning, and a rich and effective set of features for model training.
- We select and annotate a corpus of Stack Overflow posts, and use this corpus to train and test our S-NER method. We demonstrate the effectiveness of S-NER, and show that the machine learning based method can significantly outperform a well-designed rule-based baseline method. We discuss some design lessons learned in our study for researchers and designers of similar software-specific NER systems.
- We provide our annotated corpus, collected gazetteers, unsupervised word clusters, and trained CRF models to the software engineering community for further research and validation.

II. Design Challenges in NER in Software Engineering Social Content

Existing NER methods recognize real-world objects, such as person, location, organization, time and date in formal or informal texts. However, these entities are not what developers are concerned with in software engineering texts. To recognize software-specific entities (such as programming languages, platforms, frameworks, tools, libraries, APIs and software standards) that developers care about, we must develop software-specific NER methods. Little work has been done along this line, except some API extraction and linking work in software engineering texts [17], [18], [20].

In this section, we report our formative study of Stack Overflow posts to understand the challenges in designing a general NER method for software engineering social content. Understanding these design challenges helps us choose and customize state-of-the-art NER techniques for designing our software-specific NER method.

We randomly sample a diverse set of 150 Stack Overflow posts covering 6 popular programming languages (JavaScript, Java, C#, Python, PHP, HTML), 1 popular platform (Android) and 1 popular library (jQuery). We then manually identify software-specific entities mentioned in these sampled posts. Through this formative study, we summarize the following challenges in NER in software engineering social content:

1) Stack Overflow discussions are characterized by *not following strict linguistic rules* and *more spelling mistakes* compared to formal texts. For example, capitalizations are used extensively in question titles and discussions for emphasis. It happens that "JavaScript" is misspelled as "javasript" (missing the character "c").
2) Many software-specific entity names are *common words*. For example, "String" is a class name, "Application" is an Android class name, "config" can be a Python library name. However, "String", "Application" and "config" are also very common words mentioned in the discussions, for example, "this method returns string", "I am writing a Web Application", "how to config it".
3) Stack Overflow users often define code entities (e.g., classes, methods) for illustration purpose. These user-defined code elements have the *same lexical and syntactic formats* as library/framework APIs. However, they are not code entities that developers are concerned with in general.
4) Different software-specific entities often have the same name. For example, the term "memcached" can be a PHP class, and can also be a memory management tool. "Mac" can be a platform names, or a class name of Android, or an acronym of the software standard "message authentication code". This causes *ambiguity* in determining appropriate entity category.
5) The informal nature of Stack Overflow posts introduces many *name variations* for the same software-specific entity. For example, in addition to the official programming language name JavaScript, users also refer to the language as Javascript, js, javascript, or JS.
6) Different programming languages usually have different naming conventions for API entities. For example, the naming of official Java methods follows lowerCamelCase, e.g., "getProperty", PHP methods contain underscore, while .NET APIs follow UpperCamelCase.
7) Stack Overflow posts contain a plethora of distinctive named entities. Most of these entities (except for popular programming languages and platforms) are relatively infrequent.

Challenges 1-5 indicate that dictionary look-up or rule-based methods would not produce reliable NER results on software engineering social content. Furthermore, Challenges 5-7 indicate that it would be impractical or extremely expensive to define a comprehensive set of NER rules. Thus, building a machine learning based NER for software engineering social content is necessary. To tackle the above challenges, the machine learning based NER should not examine only local features of individual words. Instead, it should take into account the surrounding context of the word to recognize entity and determine entity category. Conditional Random Fields (CRFs) [21], the state-of-the-art statistical modeling methods

for solving such sequential data labeling problem, have been widely used in NLP problems, such as POS tagging [22], shallow parsing [13], and NER [23], [12], [16].

Building a machine learning based NER requires a lot of annotated data or rules for model training. Manually creating annotated data is tedious and prohibitively expensive. Furthermore, Challenges 5-6 indicate that Stack Overflow discussions contain many more Out-of-Vocabulary (OOV) words (i.e, entities that have not been seen in the training data) than formal texts, due to name variations and naming-convention differences. Challenge 7 indicates that even a large sample of manually annotated posts will still contain few training examples. Proposed solutions to alleviate this issue would be semi-supervised learning, which aims to use unsupervised word representations (e.g., Brown clusters [24]) learned from the abundant unlabeled data as extra features to improve accuracy of supervised NER model learned from small amount of annotated data [25], [13].

Due to the OOV words, it is likely to encounter an entity which is difficult to identify using local contextual cues alone because the entity has not been seen before. In these cases, a gazetteer or dictionary of known entity identifiers is often useful [26], [9], [27], [12]. The gazetteer will not be used for simple dictionary look-up. Instead, using gazetteers one may define additional features in the CRF model that represent the dependencies between a word's NER label and its presence in a particular gazetteer. Such gazetteer features are often highly informative, and including them in the model should in principle result in better model performance. Thus, we should build software-specific gazetteers for NER in software engineering social content.

III. Problem Definition

Based on our formative study, we define the problem of NER in software engineering social content as follows. Let T be a discussion thread, i.e., a question and its answers, in Stack Overflow. A question or answer is referred to as a post in Stack Overflow. Let $S \in T$ be a sentence from a Stack Overflow post. The NER task is to recognize from the sentence S a span of words $s = \langle w_1\ w_2\ ...\ w_n \rangle$ ($n \geq 1$) that refers to a software-specific named entity and classify s into the entity category it belongs to.

For any NER task, an intuitive and informative inventory of entity categories must be clearly defined. In traditional NER task, entity categories usually include person, location, organization, time and date. For NER in software engineering texts, our first task is to develop a domain-specific inventory of named entity categories which should achieve a good coverage of different aspects of software engineering knowledge that developers care about in Stack Overflow discussions.

To that end, the authors form a focus group and collaboratively review the software entities identified in our formative study. After an iterative development process, we finalize an inventory of software-specific entity categories in Table I.

We define 5 categories of software entities: *Programming Language (PL)*, *Platform (Plat)*, *API*, *Tool-library-framework (Fram)*, and *Software Standard (Stan)*. In particular, the *Programming Language* category covers different types of known programming languages, such as Object-oriented, Procedural, Scripting, Markup and Declarative. The *Platform* category refers to hardware or software platforms, such as CPU instruction sets (e.g., x86, POWER, ARMv9, Sparc), hardware architecture (e.g., CHRP, Mac), operating system and system kernel (e.g., Android, iOS). The *API* category refers to API elements of libraries and frameworks that developers can program with, such as packages, classes, interfaces, methods, functions, events and modules. The *Tool-library-framework* category broadly refers to software tools, libraries and frameworks that developers use. The *Software Standard* category refers to data formats (e.g., pdf, JSON), design patterns (e.g., Abstract Factory, Observer), protocols (e.g., HTTP), technology acronyms (e.g., Ajax), and so on.

TABLE I: Software-specific Entity Categories

Entity Category	Anno. Tag	Examples
Programming Language	PL	• Object-oriented, e.g., Java, C# • Procedural , e.g., C • Scripting, e.g., Python • Web development, e.g., JavaScript • Other types, e.g., HTML, SQL
Platform	Plat	• CPU instruction sets, e.g., x86, AMD64, MIPS • Hardware architectures, e.g., CHRP, Mac Hardware, PReP • Operating systems and system kernels, e.g., Android, Ubuntu, NT
API	API	• OOP: classes, packages, public methods, interfaces, e.g., Java ArrayList, toString() • Non-OOP: functions, routines, e.g., C malloc, printf • Others: events, built-in modules, etc. e.g., JavaScript onclick event
Tool-library-framework	Fram	• Software tools, e.g., JProfiler, Firebug, Weka • Software libraries, e.g., jQuery, NumPy, OpenCV • Frameworks, e.g., Apache Maven, Spring 4.2 • Other types of software applications, e.g., Sublime, Microsoft Word
Software Standard	Stan	• Data formats, e.g., JSON, jar, .png, .xml • Protocols, e.g., network protocols TCP, HTTP, FTP • Software design patterns, e.g., Abstract factory, Builder • Standard software technology acronyms, e.g., AJAX, JDBC

In Table II, we illustrate our software-specific NER task using some example Stack Overflow posts. Our task is to recognize and classify those software entities highlighted in boldface. Specifically, "Maven", "Mac OS X", "append", "extend", "JSON" and "Java" should be recognized as *framework*, *platform*, *API*, *API*, *software standard* and *programming language*, respectively. Note that if an entity comprises more than one word, e.g., "Mac OS X", it is recognized correctly if and only if all its words $w_1\ w_2\ ...\ w_n$ are recognized as part of the entity. Furthermore, we do not consider code elements that Stack Overflow users define to explain their questions

or answers as named entities, because these code elements are not public APIs that a community of developers are concerned with. Finally, in our NER system, we do not consider domain terminologies and concepts as named entities. For example, in the phrase "java plugin", "java" will be recognized as a programming language entity, while the domain term "plugin" is considered as a common noun, not a named entity. Similarly, we do not consider terms like "database", "sorting", "machine learning" as named entities, as they refer to general concepts, not specific entities.

TABLE II: Software-Entity Examples in Stack Overflow Posts

	Post ID	Extracted Texts
1	8826881	**Maven** Install on **Mac OS X**
2	252703	What's the difference between the list methods **append** and **extend**?
3	2591098	How to parse **JSON** in **Java**

IV. The Software-specific NER System

To address the design challenges in NER in software engineering social content, we design a semi-supervised domain-specific NER system (called S-NER) that integrates state-of-the-art supervised sequence modeling and unsupervised NLP techniques. Figure 1 shows an overview of our S-NER system. S-NER is based on Conditional Random Fields (CRF) [21] for supervised model training. S-NER utilizes a rich set of features extracted from heterogeneous data resources, including a small-sized human labeled dataset from Stack Overflow, a large-sized unlabeled dataset from Stack Overflow, and various external knowledge resources. In this section, we discuss data preparation steps, customized tokenization, human entity annotation, unsupervised word clustering, the CRF model, and our feature design for training a CRF model.

A. Data Preparation

1) Labeled Data Preparation: Supervised learning requires annotated (or labeled) data. Unlike previous studies [17], [18] that are limited to one or two programming languages, we do not restrict our NER data to be under the same programming language or platform. From Stack Overflow's official data dump released on March 16th, 2015, we randomly select posts under a diverse set of Stack Overflow tags, representing popular object-oriented and procedural languages (*java*, *c#*), Web and scripting languages (*javascript, php, python*), markup language ($html$), platform ($android$), and library ($jquery$). In fact, these 8 tags are the most frequently-used tags among all the Stack Overflow tags. Specifically, we select 1,520 Stack Overflow posts from 300 Stack Overflow discussion threads. The number of Stack Overflow posts we select for a particular Stack Overflow tag is proportional to that tag's usage frequency on Stack Overflow. We refer to this dataset as labeled data, as it will be labeled by human annotators and used for supervised learning and model testing.

We pre-process the collected posts as follows. In the official data dump of Stack Overflow, standalone code snippets are surrounded with HTML tags $\langle pre \rangle$ and $\langle /pre \rangle$. We remove these code snippets, because 1) the usage of the official APIs in such code snippets usually follows programming syntax and can be identified using rule and grammar-parser based approaches as shown in previous work [17], [18]; 2) many code elements in such code snippets are defined by question askers or answerers for illustration purpose, and these code elements do not refer to software-specific entities that other developers are concerned with. However, we keep small code elements embedded in the post texts that are surrounded with $\langle code \rangle$ and $\langle /code \rangle$ tags. These small code elements often refer to APIs, programming operators and simple user-defined code elements for explanation purpose. Removing them from the texts will impair the sentence's completeness and meaning. Finally, we strip all other HTML tags from the post texts.

2) Unlabeled Data Preparation: As mentioned in Section II, we use unlabeled Stack Overflow data to compensate for the small-sized human-labeled data. In particular, we randomly select a huge-sized data consisting of more than 7 million Stack Overflow posts from 1.8 million Stack Overflow discussion threads tagged with the 8 most frequently used tags (*java*, *c#*, *javascript*, *php*, *python*, *html*, *android*, and *jquery*). Again the number of the posts selected for a particular Stack Overflow tag is proportional to the tag's usage frequency. We refer to this dataset as unlabeled data, as it will be fed into unsupervised word clustering [24] to learn word representations (i.e., word bitstrings). The word representations will in turn be used as features for training a CRF model. Data pre-processing steps on this huge-sized unlabeled Stack Overflow texts are the same as the above steps on the labeled data.

3) External Knowledge Resources: As mentioned in Section II, gazetteers are collections of authentic named entities for a particular domain. By authentic, it means that every phrase in the gazetteer should be an entity. We can use gazetteers as features for training a CRF model. While there are many gazetteers publicly available for common person names, locations, organizations, products, temporal expressions [9], there are no gazetteers that can help to recognize software-specific entities in software engineering texts.

We contribute a set of software-specific gazetteers, including a comprehensive list of programming languages, a list of platforms, a variety of API names covering popular programming languages, a list of community-recognized software tools, libraries and frameworks, and software standards. For programming languages, we derive notable languages in existence from Wikipedia list [1]. For platforms, we obtain the gazetteer from several Wikipedia lists, including *computing platform* [2], *list of operating system* [3], *list of instruction sets* [4], *list of mobile platform* [5]. We crawl the API names as defined in Table I from the official websites of the studied programming languages (Java, JavaScript, PHP, C#, Python, HTML), platform (Android), and library (jQuery). In particular, we crawl the latest versions of

[1] https://en.wikipedia.org/wiki/List_of_programming_languages
[2] https://en.wikipedia.org/wiki/Computing_platform
[3] https://en.wikipedia.org/wiki/List_of_operating_systems
[4] https://en.wikipedia.org/wiki/List_of_instruction_sets
[5] https://en.wikipedia.org/wiki/List_of_mobile_software_distribution_platforms

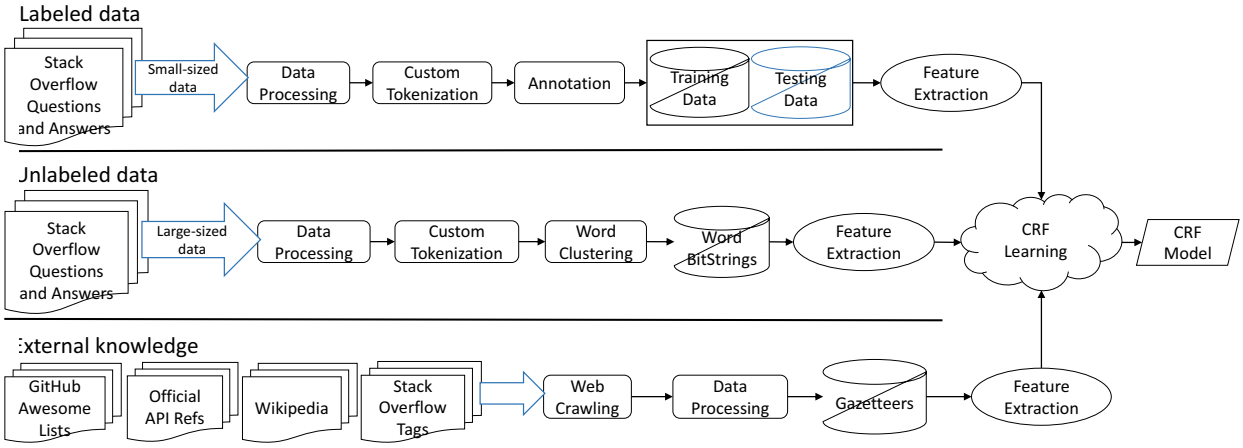

Fig. 1: S-NER System Working Flow

the APIs for Java 8, PHP 5, Android API level 23. For C#, we collect its API names for .NET 4.5 and 4.6. For Python, we crawl its modules and methods for version 2.7 and 3.4. For HTML, we collect both HTML tags and DOM methods. For jQuery, we collect methods for the jQuery library and the jQuery UI library. For software tools, libraries and frameworks of different programming languages and platforms, we obtain the list from GitHub Awesome Lists. GitHub Awesome Lists is a popular software knowledge source on GitHub curated by developers. For every main-stream programming language or platform, there exists a curated list of well-known software tools, libraries and frameworks, as well as other types of information for that language or platform (See more at: https://github.com/sindresorhus/awesome). For software standards, we obtain design pattern and protocol names from Wikipedia lists. We cannot show all the data sources here due to space limitation. We collect a list of data formats and technology acronyms from Stack Overflow tags.

TABLE III: An Example of Our S-NER's Tokenization

Input Sentence	What's␣the␣equivalent␣of␣java's␣Thread.sleep()␣in␣js?
Stanford Tokenizer	What␣'s the␣equivalent␣of␣java␣'s␣Thread␣.␣sleep␣(␣)␣in␣js␣?
S-NER Tokenization	What␣'␣s␣the␣equivalent␣of␣java␣'␣s␣Thread.sleep()␣in␣js␣?

B. Customized Tokenization

Tokenizers designed for general English texts cannot properly handle software engineering social content which is both social and technical. We develop a domain-specific tokenizer for handling texts with software-specific entities. The tokenizer uses regular expressions to match valid URLs, at-mentions, and emoticons (e.g., :), :))). The tokenizer does not split the name of a software entity, e.g., the name of an API. It does not split valid programming operators, such as "==" and "!=". It considers separate parentheses, i.e., '(' and ')', as punctuations. However, parentheses, as well as dot, #, and $, that appear in an API are considered as part of the API itself. In Table III, we show an example of the S-NER's tokenization results. Line 1 is the input sentence. Line 2 is the tokenization done by Stanford Tokenizer, which is designed for general English texts. Line 3 is tokenized by S-NER. As we can see, S-NER is able to tokenize the Java API *Thread.sleep()* as a whole, while the tokenizer for general texts splits the API name into 5 tokens.

C. Human Entity Annotation

For annotation, we use Brat [28], a web-based annotation tool. We adopt the widely used BIO representation of text chunks. In our context, BIO means the Begin, Inside and Outside of an entity. Take the 5-token sentence "*Apache ant is a tool*" as an example. The correct annotation is "*B-Fram I-Fram O O O*", where "*Fram*" is the annotation tag as shown in Table I, and B and I indicate the Begin and Inside of the text chunk. This means that the phrase "*apache ant*" refers to a framework, while the last three words are not entities.

The annotation process involves 3 stages and is performed by 9 annotators who are all from computer science background with 5+ years of programming experience. Before annotation, we give all annotators an 1-hour tutorial regarding the tool usage, annotation methods, and entity categories. We provide some annotation examples for the annotators to practice. The purpose is to let them reach a consensus on what kinds of entities to annotate and how.

In Stage 1, each annotator is assigned with some Stack Overflow posts. During this manual annotation process, we ask them to report to us when there are tokenization errors or deficiencies, and when certain tokens are hard to be labeled using the software-specific entity categories we develop. After this stage, we use the feedbacks from our annotators to improve the tokenization, refine our software entity categories, and clean up the annotated data. In Stage 2, we let annotators cross validate the data, i.e., the same set of tokens from Stage 1 is examined by a different annotator in Stage 2. In Stage 3, a final sweep to all the annotated data is made by the first, third and forth author of this paper to improve the consistency of our annotation.

D. Unsupervised Word Clustering

To alleviate the problem of out-of-vocabulary (OOV) and lexical word variations, we rely on unsupervised word clustering to group together words that are distributionally similar. Specifically, we apply Brown Clustering [24], [29] on the unlabeled Stack Overflow posts (see Section IV-A2). Brown Clustering assigns words that appear in similar contexts into the same cluster. Words in the cluster are represented as a bitstring. We use Liang's implementation of Brown Clustering [6]. We configure the number of clusters to 1000 and we only cluster words that appear no less than 10 times. It takes 15 hours to finish the word clustering of the unlabeled dataset on a 4-core Intel i5-4570 processor. Table IV lists some resulting word clusters and their corresponding bistrings. We can see that word clusters can represent semantically similar words in Stack Overflow posts.

TABLE IV: Example Word Clustering Results

Bitstring	Top words (by frequency)
11111011110	.NET Spring ASP.NET Django HTML5 asp.net bootstrap django Bootstrap Entity spring .Net .net wordpress Wordpress Angular AngularJS JPA
11111111110	foreach setTimeout setInterval eval Files json_encode explode exec var_dump print_r await document.write
111111111111111	hover touch mouseover blur keyup keypress keydown mouseout fadeIn mouseenter mouseleave mousedown delegated show() fadeOut mouseup mousemove hide()

We also build an HTML viewer for interested readers to browse and check our detailed word clustering results. We host the HTML viewer at this web service: http://cyong.oneinfinityloop.com/clusters/cluster_viewer.html.

E. Supervised Learning based on CRF

S-NER is based on linear chain Conditional Random Fields (CRF). We describe the CRF model here and the features we use to train the CRF model.

1) Model: Given a sequence of observations (tokens in this work) $\vec{x} = x_1, x_2, ..., x_n$, we want to assign each observation with a label (from our annotation tags shown in Table I and BIO representation of text chunks), e.g., B-API, I-API, B-Plat, I-Plat, O, and so on. This sequence of labeling is denoted as $\vec{y} = y_1, y_2, ..., y_n$. In linear chain CRF, the probability vector of assigning $\vec{y}$ based on $\vec{x}$ is:

$$p_{\vec{\lambda}}(\vec{y}|\vec{x}) \propto \exp(\sum_{j=1}^{n}\sum_{i=1}^{m}\lambda_i f_i(y_{j-1}, y_j, \vec{x}, j)) \quad (1)$$

where j specifies the position in the input sequence $\vec{x}$, $\vec{f}(y_{j-1}, y_j, \vec{x}, j)$ are the features to be designed for training and testing, and λ_i represents the weight of feature f_i.

2) Feature Design: We extract a rich set of features from annotated corpus, unlabeled Stack Overflow texts, and external knowledge resources.

Orthographic features: We design the following orthographic features based on our observations from Stack Overflow texts. We first use regular expressions to detect URLs, at-mentions and emoticons. We consider as features whether a token is initial capitalized, whether all characters in a token are capitalized, whether a token is alphanumeric, whether it contains digits, underscores, or dots. We further examine if a token has parentheses at the end, whether a token contains both digits and dots, whether a token has capitalizations in the middle. If a token has a dot, we also check if its suffixes match a data format name in our collected "software standard" gazetteer. We do so because many files names, e.g., "MyCode.java", contain a dot but they are not named entities. Note that many URLs have the above mentioned features, therefore, we normalize a URL into "@u@" once it is detected using regular expressions.

Lexical and contextual features: We consider every token in our annotated corpus as a feature. We also consider the uppercase form and the lowercase form of every token as features. To utilize the context information, a window size [-2, 2] is used to add the previous and the next two tokens as features. We experiment other window size settings, but find them not helpful for performance improvement.

Word bitstring features: As mentioned in Section IV-D, a word is represented as a bitstring after word clustering. We use prefixes of the bitstrings as features. Based on our word clustering results, the prefix lengths we use are $5, 6, 7, 8, 9, ..., 15$. Take the bitstring at line 1 of Table IV as an example. The length of this bitstring is 11. For prefixes lengths 5-11, the prefixes used as features are "11111", "111110", ..., "11111011110". For prefix lengths 12-15, the whole bitstring is used as features.

Gazetteer features: We store the gazetteers for different entity categories into different files. For each programming language, we store the names of packages, classes, modules, methods, events, etc., in separate files. We use string matching results against gazetteer entries in different gazetteer files as features. Specifically, we perform exact string matching for class names and module names. We perform lowercase string matching for entries that only have one word. We perform fuzzy string matching using the *fuzzywuzzy* tool [7] for entries that consist of a span of words. If the fuzzy ratio is above 0.8, we consider it as a match.

V. Evaluation

Our experiments are designed to demonstrate the need of a machine learning based software-specific NER system, and to test the efficacy of the software-specific feature set we develop, given a small-sized annotated software engineering corpus.

A. Experimental Setup

The annotated corpus consists of 4,646 sentences derived from 1,520 Stack Overflow posts. The total number of tokens after tokenization is 70,570. The number of software-specific named entities is 2,404. Stack Overflow discussions are entity-rich, as evidenced by the fact that there are on average 1.58 (2404/1520) software entities per Stack Overflow post, according to our annotation results.

[6]https://github.com/percyliang/brown-cluster

[7]https://github.com/seatgeek/fuzzywuzzy

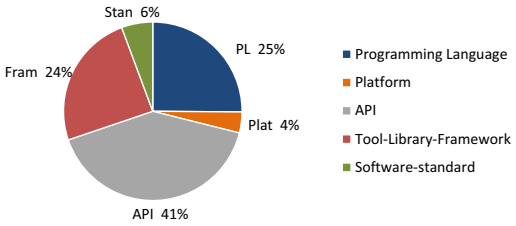

Fig. 2: Proportion of Different Categories of Software-specific Named Entities in Our Annotated Corpus

In Figure 2, we further show the proportions of different categories of software-specific entities according to our predefined entity categories in Table I. We can see that API is the most discussed entity category among developers, which accounts for 41% of all software-specific entities in our corpus.

For model training and testing, we use 10-fold cross validation. We randomly divide our annotated corpus into 10 equal-sized subsets. Of the 10 subsets, one single subset is retained as the testing data, and the remaining 9 subsets are used as training data. We repeat this process 10 times and produce a single estimation by averaging the 10 results obtained.

It is very likely that a question and its answers discuss the same set of software entities. Therefore, to avoid model overfitting, we make sure that the answers to a particular question will not be put in the testing data if the corresponding question is in the training data.

For the implementation of linear chain CRF, we use CRF++ [8], a popular CRF toolkit that has been widely used for sequential tagging tasks like NER.

B. Evaluation Metrics

We use standard NER evaluation metrics, i.e., precision, recall, and F1. For each category of named entity, precision measures what percentage the output labels are correct. Recall measures what percentage the named entities in the golden-dataset are labeled correctly. F1-score is the harmonic mean of precision and recall.

Using 10-fold cross validation produces 10 sets of testing results. We calculate the average precision, recall and F1-score as the overall performance of our S-NER system. We report phrase-level precision, recall and F1. This means that if an entity consists of a span of tokens, it is considered correctly labeled if and only if all its tokens are labeled correctly (see that example of "Mac OS X" in Table II in Section III).

C. Baseline System

Our baseline system is implemented using a mixture of empirical lexical rules and dictionary look-ups. The dictionary look-up is based on the gazetteers we collect from external knowledge resources (see Section IV-A3). The testing data used to evaluate the baseline system and the CRF model are always the same.

Since our gazetteers cover a very broad range of software-specific entities of different categories, we initially use these gazetteers directly for string matching. If a span of tokens matches an entry in the gazetteer, we label it as an entity of the corresponding category. However, this simple string-matching approach performs poorly. For example, the F1 of recognizing *Programming Language* entities and *Tool-library-framework* entities are as low as 45% and 17%, respectively. Our error analysis indicates that:

- Many entries in the gazetteers are common words, or have only one single character. To name a few, "B", "D", "Go", "GOAL", "Logo" are programming language names. "Application" is an Android class. "Moment [9]" is a JavaScript date library, "click [10]" is a Python library, "Task [11]" is a tool for running PHP tasks. Such common-words or single-character entity names can impair the system performance significantly when using string matching. This is not a common phenomena for real-word objects, such as person, location, and organization.
- Direct string matching is unable to handle name variations. For example, some users write the Android class "ListView" as "List View", "listview", etc.
- For API names, users sometimes follow the standard format *package.class.method* or *class.method*, but sometimes write the method names directly.

To improve the performance of the baseline system, we further analyze the gazetteers of different entity categories and design some lexical rules as follows.

- For the gazetteer of programming language entities, we choose not to use the comprehensive list of programming languages which consists of 419 programming languages. Rather, we manually identify programming languages from Stack Overflow tags with tag frequency greater than 10,000. We compile a short list of 30 popular languages, and we add some of the commonly seen lexical variations, e.g., "js" for "JavaScript".
- We identify class and method names that are not compound words and remove them from the gazetteer. Here compound words refer to words like "ListView" which is made of "List" and "View". Some examples of the removed APIs include: "Application" Class in Android, "Array" Class in Java, etc. We store packages, classes, methods for a certain language as separate entries in the gazetteers, and use them separately or combine them as necessary to match API mentions. Notice that formally written APIs have distinguishable orthographic features, e.g., formal PHP methods contain underscore in the middle and parentheses at the end, some Python methods follow the syntax of *Module.Method()*. Therefore, we add regular expressions to detect those formal APIs.
- For the tool-library-framework gazetteers from GitHub Awesome Lists, we manually identity those that can not be differentiated from common words, such as the above mentioned library and tool names "Moment", "click",

[8] https://taku910.github.io/crfpp/

[9] http://momentjs.com/

[10] http://click.pocoo.org/5/

[11] http://taskphp.github.io/

"Task". We remove a list of such tool-library-frameworks from the gazetteers (we do not list all of them here due to space limitation).

- If a token begins with a dot followed by an entry in the software standard gazetteer, we label it as a software standard entity. For example, ".jar" and ".pdf" are labeled as software entities.
- The string matching methods against gazetteer entities are similar to what we do to the gazetteer feature design (see Section IV-E2).
- We further add some empirical rules. If the current word matches a one-word entry in the programming language, or the platform, or the tool-library-framework gazetteer, we check if its next word is made of digits and dots, we also check if its previous word is a software company or organization name (we manually identity a list of software organization names, such as microsoft, apache, etc.). We do so to enhance the detection of software entities that consist of span of words, such as "python 2.7", "apache ant", "microsoft excel", etc.

Our experience with the baseline system suggests that it is not an easy task to design a robust dictionary and rule based NER system for software engineering social content. We invest a significant effort to improve the baseline so as to make a fair comparison with the machine learning based NER system.

D. Overall Comparison Results of All Entity Categories

In Table V, we show the overall results when using S-NER and the baseline system to recognize all 5 categories of software-specific entities defined in Table I. We see that the overall F1-score of S-NER is 78.176%, which outperforms that of the baseline system by 30.3%. The improvements of precision and recall when comparing S-NER against the baseline system are 47% and 14.4%, respectively.

TABLE V: Overall Experimental Results

System	Precision(%)	Recall(%)	F1(%)
Baseline	55.849	65.293	60.018
S-NER	**82.093**	**74.706**	**78.176**
	47.0% ↑	14.4% ↑	30.3% ↑

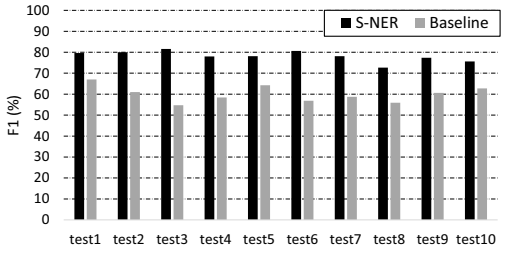

Fig. 3: Comparison of S-NER and the Baseline System for the 10 Testing Sets

Recall that we use 10-fold cross validation. In Figure 3, we show the detailed comparisons between S-NER and the baseline on the 10 testing data. We observe that the performance of S-NER exceeds its baseline system consistently on all the testing data. The largest improvement among all the test data occurs in test3, the F1-score of S-NER is slightly more than 81%, while the F1 of the baseline system is around 55%. 81% is the highest F1 obtained for S-NER (test3), while the lowest F1 of S-NER is about 73% (test8). The highest and lowest F1 of the baseline system are 68% (test1) and 55% (test3).

E. Comparison Results of Individual Entity Category

We report the experimental results of each individual entity category in Table VI. As we can see, the NER performances of both S-NER and the baseline system on different categories of entities are quite different.

TABLE VI: NER Results of Individual Entity Category

(a) Programming Language Category

System	Precision(%)	Recall(%)	F1(%)
Baseline	87.918	87.548	87.457
S-NER	92.146	94.674	93.358
	4.8% ↑	8.1% ↑	6.7% ↑

(b) Platform Category

System	Precision(%)	Recall(%)	F1(%)
Baseline	75.447	80.506	76.423
S-NER	79.979	76.938	77.204
	6.0% ↑	4.4% ↓	1.0% ↑

(c) API Category

System	Precision(%)	Recall(%)	F1(%)
Baseline	45.336	66.343	53.105
S-NER	77.515	66.521	71.018
	71.1% ↑	0.3% ↑	33.7% ↑

(d) Tool-library-framework Category

System	Precision(%)	Recall(%)	F1(%)
Baseline	47.565	42.967	44.47
S-NER	74.805	69.019	71.702
	57.1% ↑	60.5% ↑	61.1% ↑

(e) Software Standard Category

System	Precision(%)	Recall(%)	F1(%)
Baseline	54.341	62.121	55.891
S-NER	88.117	74.686	79.995
	62.2% ↑	20.1% ↑	43.1% ↑

It follows our expectation that the NER of programming languages achieves very high precision, recall and F1 for both S-NER and the baseline, as shown in Table VIa. The rule and dictionary look-up based baseline is able to perform well with F1 at 87.457%. One example error made by the baseline system is that it excessively labels the word "C" as programming language, even though many times "C" is just a common char. By comparison, the machine learning based S-NER considers the contextual environment of the current word, as covered in Section IV-E2, and is more robust (93.358% F1).

In Table VIb, we show the results for the NER of platforms. We can see that the F1 of the baseline system is very close to that of S-NER. S-NER has the better precision, but the baseline

system has the better recall. We can understand this result from two aspects. First, the naming of platforms is not as ambiguous as the naming of other categories of entities. Common words are not frequently used as platform names. Second, the number of known platforms is much less compared to the number of APIs, tools, frameworks, etc. These factors reduce the difficulty of platform NER.

From Table VIc, VId, VIe, we see that the machine learning based S-NER can outperform the baseline significantly when recognizing APIs, tool-library-frameworks, and software standards. The F1 improvements observed for these three entity categories are all higher than the overall improvement shown in Table V. These results suggest that it can be very difficult to develop a robust rule based approach to recognize the informally written API names, tool-library-frameworks, and software standards in software engineering social content like Stack Overflow, because it is impossible to know in advance what word variations and ambiguities there would be.

F. Feature Ablation

We also perform independent feature ablation experiments to study the effect of individual feature(s) on the NER performance. In Table VII, we ablate one kind of feature(s) at a time from our full feature set and test the resulting F1-score.

We find that using unsupervised word bitstrings as features is very helpful, without which the overall F1 drops sharply from 78.176% to 72.642%. The dropping of F1 without word clustering features is the most significant for the API category and the tool-library-framework category.

The use of gazetteers as features has small impact on the final system performance. Removing gazetteer features only leads to the decrease of F1 score for about 0.5%. This is noteworthy: gazetteer features are considered critical in other NER work [9], [12], especially for NER in social texts such as Tweets [12]. In these studies, gazetteers can boost the NER performance tremendously by an F1 increase of 19% as reported in [12]. We will discuss more about the design of software-specific gazetteers in Section VI.

Our results show that initial capitalization feature is not particularly useful for software-specific NER, as we raised in Section II. The F1 without initial capitalization is slightly lower at 77.577%. Without prefixes and suffixes features, F1 decreases slightly to 76.719%. We also ablate other orthographic features one by one, and we find that the absence of one particular orthographic feature does not significantly impair S-NER's F1 score. Overall, S-NER's performance is contributed by the combined action of all its features.

G. Varying Labeled Data Size

We want to further understand how S-NER's performance changes with the size of the labeled data, so as to know how much data we should label to reach a reasonable F1-score. In this set of experiments, we turn on our full-feature set during training and testing. We randomly select one-tenth, two-tenth, three-tenth, ..., nine-tenth and all of the original labeled dataset, and use these datasets for model training and testing. For each dataset, we use 10-fold cross validation. We report the corresponding averaged F1 of S-NER in Figure 4. As we increase the size of labeled data, we see F1 increases monotonically. The smaller the size of the labeled data, the larger the increase rate. The F1-score becomes relatively stable after we use about 80% of all the labeled data.

TABLE VII: The Effects of Individual Feature(s)

	F1-score for each entity category (%)					**Overall**
	PL	Plat	API	Fram	Stan	**F1(%)**
Full-feature	93.358	77.204	71.018	71.702	79.995	78.176
w/o word clustering	91.327	77.492	66.507	57.581	75.823	72.642
w/o gazetteer	92.395	75.092	70.372	72.197	78.476	77.691
w/o affixes features	92.268	76.642	69.282	71.34	76.901	76.719
w/o init. Capital.	91.862	79.982	69.779	72.801	79.417	77.577

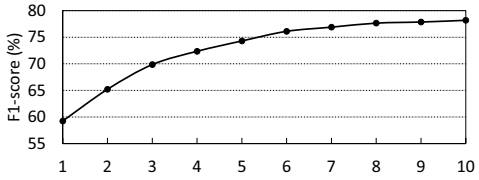

Fig. 4: Effects of Labeled Data Size on S-NER's Performance

VI. Why Does Gazetteer Not Work?

Two important decisions in the design of our S-NER system are to include unsupervised word representations and gazetteers as features for training the CRF model. As reported in Table VII, unsupervised word clustering boosts the NER performance as expected. However, gazetteers have marginal effect on the performance of S-NER. This contradicts the results of many NER studies showing that insertion of gazetteers as features in machine learning based NER can significantly boost NER performance [26], [9], [27], [12].

The principle of how gazetteer features work is that: if a span of tokens (n$\geq$1) in the training dataset matches a gazetteer entry and is labeled as an entity, the CRF model will know that a string matching in that gazetteer is an indicator of entity appearance. Accordingly, the weight λ (see Eq.1) to that specific gazetteer feature will increase. However, software-specific gazetteers contain highly common words. Take the Android class-name gazetteer as an example. Words like "Application", "Path", "Array" are Android classes. In most of the situations, these words are labeled as "O", i.e., they are not entities in Stack Overflow texts. Therefore, the weight λ of such gazetteer features will decrease during CRF model training. As a result, even if in some cases that these common words indeed refer to software entities, it could be difficult for S-NER to recognize them.

A common practice to alleviate the issue of common words in NER tasks is to remove these common words from the gazetteer [26], [27], similar to what we do for the baseline system, as covered in Section V-C. However, this practice proves to be not very helpful for software-specific NER. We only observe a small performance improvement for the recognition of tool-library-frameworks after we remove common words from gazetteers. The recognition of APIs, which occupy the

largest proportion of all entities, has almost no improvements. Again, we use the example of Android class-name gazetteer to understand this observation. After removing common words, the entries left in the Android class gazetteer are mostly compound words, such as "AbsListView", "AbstractList" and "AbsListView.LayoutParams". These words have distinct orthographic features, e.g., their suffixes, prefixes and capitalization patterns. Words with similar orthographic features appear frequently in the training dataset. Consequently, even without the insertion of gazetteers, the CRF model can recognize these words accurately. As such, gazetteers have little impact on the NER performance.

Our analysis indicates that: how to design a high-quality gazetteer for software-specific NER remains to be an open question for the software engineering research community.

VII. Related Work

A. Software Engineering Information Extraction and Linking

In software engineering community, software information extraction and linking have been extensively studied.

A large body of the work focus on code element extraction and linking [17], [18], [11], [30], [31], [32], [33], [34], [35], [19]. In this line of research, the named entities being recognized involve and only involve APIs of certain programming languages, usually formulated as a traceability recovery problem. For example, Rigby and Robillard [17] develop a code element extraction and linking tool and propose the notion of code salience as an indicator of the importance of a particular code element. Subramanian et al. [18] build a browser extension that links an API in code snippets of Stack Overflow to its corresponding API reference documentation.

Some work recognizes important words or concepts in software artifacts to facilitate content comprehension [20], [36], [37]. For example, Shokripour et al. [36] use part-of-speech information to find software noun terms in bug reports. Hauff et al. [37] utilize the DBPedia Ontology to extract software concepts from GitHub developer profiles.

Other related work includes: Witte et al. [38] build ontology representations for software artifacts. In [39], natural language parsing is used to classify the content of development emails. Bagheri and Ensan [40] mine Wikipedia contents and recommend Wikipedia entries as potential tags for tagging Stack Overflow posts. Sharma et al. [41] identify Tweets that contain software engineering knowledge using language model. Tian et al. [42], Yang and Tan [43], [44] and Howard et al. [45] study software-specific word similarity in software texts.

Comparing with these studies, our work aims to recognize a wide range of software-specific entities (not limited to code elements of a particular programming language), and classify each recognized entity into the entity category it belongs to. In particular, we formulate the research problem of NER in software engineering and present a working solution.

B. Named Entity Recognition in Other Domains

One representative work of Traditional NER is done by Ratinov and Roth [9]. They systematically study the design challenges of NER in formal English texts, and point out design decisions should be made in traditional NER.

Apart from formal English texts, NER has also been widely studied in other genres of texts.

NER in social media. The informal nature of social texts introduces new challenges to NER. Liu et al. [12] recognize entities from Tweets, and report an average F1 of 80% when recognizing persons, locations and organizations. Ritter et al. [13] also investigate NER in Tweets. They recognize a wider range of named entities and achieve an overall F1 of 66%.

NER in bioinformatics. Biomedical-specific named entity recognition (Bio-NER) is an another active research field. Bio-NER recognizes bio-specific entities such as protein, DNA, RNA and cell. Bio-NER has been raised as a community task [14]. Machine learning based systems are commonly used and are found to outperform rule based systems [15], [16].

Commercial products NER. Yao and Sun [23] perform mobile phone names recognition and normalization in Internet forums. Wu et al. [46] recognize mentions of consumer products from user-generated comments on the Web.

NER in non-English natural language. Wu et al. [47] propose a customized statistical model for NER in Chinese. Shaalan and Raza [48] develop a NER system for Arabic.

VIII. Conclusion and Future Work

In this paper, we formulate the research problem of NER in software engineering. To design a software-specific NER system, we show that one must first understand the unique characteristics of domain-specific texts that bring unique design challenges. Then, based on the understandings of these design challenges, we show how we combine state-of-the-art supervised and unsupervised machine learning and NLP techniques to design an effective software-specific NER solution, which can reduce the demand for labeled data, meanwhile maintain the generality and robustness of the NER system.

We build S-NER, a semi-supervised machine learning method for NER in software engineering social content, and demonstrate that S-NER significantly outperforms a well-designed rule-based NER system when applied on Stack Overflow posts. In the process of building this NER system, we contribute an inventory of software-specific entity categories, a corpus of labeled Stack Overflow posts, a software-specific tokenizer, a collection of software-specific gazetteers, unsupervised word clusters, and a rich and effective set of features for NER in software engineering texts. We release our annotated dataset and trained CRF models [12] for community validation and further research.

The method presented in this paper can be extended to more software engineering texts. We are on a continuous effort to extract software-specific entities from different types of software engineering texts (e.g., API documentations, bug reports, Tweets), and to develop entity-centric search systems for the software engineering domain.

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

[12] https://drive.google.com/open?id=0ByoLWPpAxGVFdEROT09EMC0zUzg

REFERENCES

[1] L. Mamykina, B. Manoim, M. Mittal, G. Hripcsak, and B. Hartmann, "Design lessons from the fastest q&a site in the west," in *Proceedings of the SIGCHI conference on Human factors in computing systems*. ACM, 2011, pp. 2857–2866.

[2] C. Parnin, C. Treude, L. Grammel, and M.-A. Storey, "Crowd documentation: Exploring the coverage and the dynamics of api discussions on stack overflow," *Georgia Institute of Technology, Tech. Rep*, 2012.

[3] D. M. Blei, A. Y. Ng, and M. I. Jordan, "Latent dirichlet allocation," *the Journal of machine Learning research*, vol. 3, pp. 993–1022, 2003.

[4] G. Zhou, T. He, J. Zhao, and P. Hu, "Learning continuous word embedding with metadata for question retrieval in community question answering," in *Proceedings of the 53rd Annual Meeting of the Association for Computational Linguistics and the 7th International Joint Conference on Natural Language Processing (Volume 1: Long Papers)*. Association for Computational Linguistics Beijing, China, 2015, pp. 250–259.

[5] T. Lin, P. Pantel, M. Gamon, A. Kannan, and A. Fuxman, "Active objects: Actions for entity-centric search," in *Proceedings of the 21st international conference on World Wide Web*. ACM, 2012, pp. 589–598.

[6] M. J. Carman, F. Crestani, M. Harvey, and M. Baillie, "Towards query log based personalization using topic models," in *Proceedings of the 19th ACM international conference on Information and knowledge management*. ACM, 2010, pp. 1849–1852.

[7] E. Meij, K. Balog, and D. Odijk, "Entity linking and retrieval for semantic search." in *WSDM*, 2014, pp. 683–684.

[8] P. Pantel and A. Fuxman, "Jigs and lures: Associating web queries with structured entities," in *Proceedings of the 49th Annual Meeting of the Association for Computational Linguistics: Human Language Technologies-Volume 1*. Association for Computational Linguistics, 2011, pp. 83–92.

[9] L. Ratinov and D. Roth, "Design challenges and misconceptions in named entity recognition," in *Proceedings of the Thirteenth Conference on Computational Natural Language Learning*. Association for Computational Linguistics, 2009, pp. 147–155.

[10] E. Minkov, R. C. Wang, and W. W. Cohen, "Extracting personal names from email: Applying named entity recognition to informal text," in *Proceedings of the conference on Human Language Technology and Empirical Methods in Natural Language Processing*. Association for Computational Linguistics, 2005, pp. 443–450.

[11] 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.

[12] 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.

[13] A. Ritter, S. Clark, O. Etzioni *et al.*, "Named entity recognition in tweets: an experimental study," in *Proceedings of the Conference on Empirical Methods in Natural Language Processing*. Association for Computational Linguistics, 2011, pp. 1524–1534.

[14] J.-D. Kim, T. Ohta, Y. Tsuruoka, Y. Tateisi, and N. Collier, "Introduction to the bio-entity recognition task at jnlpba," in *Proceedings of the International Joint Workshop on Natural Language Processing in Biomedicine and Its Applications*, ser. JNLPBA '04. Association for Computational Linguistics, 2004, pp. 70–75.

[15] K. Yoshida and J. Tsujii, "Reranking for biomedical named-entity recognition," in *Proceedings of the Workshop on BioNLP 2007: Biological, Translational, and Clinical Language Processing*. Association for Computational Linguistics, 2007, pp. 209–216.

[16] Y. Wang, "Annotating and recognising named entities in clinical notes," in *Proceedings of the ACL-IJCNLP 2009 Student Research Workshop*. Association for Computational Linguistics, 2009, pp. 18–26.

[17] 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.

[18] 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.

[19] S. Gupta, S. Malik, L. Pollock, and K. Vijay-Shanker, "Part-of-speech tagging of program identifiers for improved text-based software engineering tools," in *Program Comprehension (ICPC), 2013 IEEE 21st International Conference on*. IEEE, 2013, pp. 3–12.

[20] G. Capobianco, A. D. Lucia, R. Oliveto, A. Panichella, and S. Panichella, "Improving ir-based traceability recovery via noun-based indexing of software artifacts," *Journal of Software: Evolution and Process*, vol. 25, no. 7, pp. 743–762, 2013.

[21] 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.

[22] K. Gimpel, N. Schneider, B. O'Connor, D. Das, D. Mills, J. Eisenstein, M. Heilman, D. Yogatama, J. Flanigan, and N. A. Smith, "Part-of-speech tagging for twitter: Annotation, features, and experiments," in *Proceedings of the 49th Annual Meeting of the Association for Computational Linguistics: Human Language Technologies: short papers-Volume 2*. Association for Computational Linguistics, 2011, pp. 42–47.

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

[24] 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*, pp. 467–479, 1992.

[25] 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.

[26] A. Toral and R. Munoz, "A proposal to automatically build and maintain gazetteers for named entity recognition by using wikipedia," in *Proceedings of EACL*, 2006, pp. 56–61.

[27] J. Kazama and K. Torisawa, "Exploiting wikipedia as external knowledge for named entity recognition," in *Proceedings of the 2007 Joint Conference on Empirical Methods in Natural Language Processing and Computational Natural Language Learning (EMNLP-CoNLL)*, 2007, pp. 698–707.

[28] P. Stenetorp, S. Pyysalo, G. Topić, T. Ohta, S. Ananiadou, and J. Tsujii, "brat: a web-based tool for NLP-assisted text annotation," in *Proceedings of the Demonstrations Session at EACL 2012*. Avignon, France: Association for Computational Linguistics, April 2012.

[29] P. Liang, "Semi-supervised learning for natural language," Ph.D. dissertation, Massachusetts Institute of Technology, 2005.

[30] 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.

[31] 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.

[32] B. Dagenais and H. Ossher, "Automatically locating framework extension examples," in *Proceedings of the 16th ACM SIGSOFT International Symposium on Foundations of software engineering*. ACM, 2008, pp. 203–213.

[33] 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.

[34] A. De Lucia, R. Oliveto, and G. Tortora, "Adams re-trace: traceability link recovery via latent semantic indexing," in *Proceedings of the 30th international conference on Software engineering*. ACM, 2008, pp. 839–842.

[35] J.-R. Falleri, M. Huchard, M. Lafourcade, C. Nebut, V. Prince, and M. Dao, "Automatic extraction of a wordnet-like identifier network from software," in *Program Comprehension (ICPC), 2010 IEEE 18th International Conference on*. IEEE, 2010, pp. 4–13.

[36] R. Shokripour, J. Anvik, Z. M. Kasirun, and S. Zamani, "Why so complicated? simple term filtering and weighting for location-based bug report assignment recommendation," in *Proceedings of the 10th Working Conference on Mining Software Repositories*. IEEE Press, 2013, pp. 2–11.

[37] C. Hauff and G. Gousios, "Matching github developer profiles to job advertisements," in *Proceedings of the 12th Working Conference on Mining Software Repositories*, ser. MSR '15. IEEE Press, 2015, pp. 362–366.

[38] R. Witte, Y. Zhang, and J. Rilling, "Empowering software maintainers with semantic web technologies," in *The Semantic Web: Research and Applications*. Springer, 2007, pp. 37–52.

[39] A. Di Sorbo, S. Panichella, C. Visaggio, M. Di Penta, G. Canfora, and H. Gall, "Development emails content analyzer: Intention mining in developer discussions," in *30th international conference on Automated Software Engineering (ASE 2015). Lincoln, Nebraska*, 2015.

[40] E. Bagheri and F. Ensan, "Semantic tagging and linking of software engineering social content," *Automated Software Engineering*, pp. 1–44, 2014.

[41] A. Sharma, Y. Tian, and D. Lo, "Nirmal: Automatic identification of software relevant tweets leveraging language model," in *Software Analysis, Evolution and Reengineering (SANER), 2015 IEEE 22nd International Conference on*. IEEE, 2015, pp. 449–458.

[42] Y. Tian, D. Lo, and J. Lawall, "Automated construction of a software-specific word similarity database," in *Software Maintenance, Reengineering and Reverse Engineering (CSMR-WCRE), 2014 Software Evolution Week-IEEE Conference on*. IEEE, 2014, pp. 44–53.

[43] J. Yang and L. T. , "Inferring semantically related words from software context," in *Proceedings of the Working Conference on Mining Software Repositories (MSR'12)*, June 2012.

[44] J. Yang and L. Tan, "Swordnet: Inferring semantically related words from software context," *Empirical Software Engineering*, vol. 19, no. 6, pp. 1856–1886, 2014.

[45] M. J. Howard, S. Gupta, L. Pollock, and K. Vijay-Shanker, "Automatically mining software-based, semantically-similar words from comment-code mappings," in *Proceedings of the 10th Working Conference on Mining Software Repositories*. IEEE Press, 2013, pp. 377–386.

[46] S. Wu, Z. Fang, and J. Tang, "Accurate product name recognition from user generated content," in *Data Mining Workshops (ICDMW), 2012 IEEE 12th International Conference on*. IEEE, 2012, pp. 874–877.

[47] Y. Wu, J. Zhao, and B. Xu, "Chinese named entity recognition combining a statistical model with human knowledge," in *Proceedings of the ACL 2003 workshop on Multilingual and mixed-language named entity recognition-Volume 15*. Association for Computational Linguistics, 2003, pp. 65–72.

[48] K. Shaalan and H. Raza, "Nera: Named entity recognition for arabic," *Journal of the American Society for Information Science and Technology*, vol. 60, no. 8, pp. 1652–1663, 2009.