On domain knowledge organization and extraction in software engineering

Ye, Deheng

2017

Ye, D. (2017). On domain knowledge organization and extraction in software engineering.
Doctoral thesis, Nanyang Technological University, Singapore.

<http://hdl.handle.net/10356/69477>

<https://doi.org/10.32657/10356/69477>

**NANYANG
TECHNOLOGICAL
UNIVERSITY**

**ON DOMAIN KNOWLEDGE
ORGANIZATION AND EXTRACTION IN
SOFTWARE ENGINEERING**

DEHENG YE

SCHOOL OF COMPUTER SCIENCE AND ENGINEERING

2017

**ON DOMAIN KNOWLEDGE
ORGANIZATION AND EXTRACTION IN
SOFTWARE ENGINEERING**

DEHENG YE

School of Computer Science and Engineering

A thesis submitted to the Nanyang Technological University
in partial fulfillment of the requirement for the degree of
Doctor of Philosophy

2017

Abstract

Developers' social information seeking on the Web is unable to benefit from the recent significant advances of semantics-oriented applications, such as knowledge graph and direct answers. This is largely because existing approaches to analyzing software engineering social content, such as the discussions on Stack Overflow, 1) treat software-specific entities in the same way as other textual content, and 2) fall short to consider the semantic linkages between software knowledge. In this thesis, we perform a pioneering study towards the long-term goal of enabling domain-specific knowledge graph and semantic search in software engineering. Using the developer-generated content on Stack Overflow, we formulate a series of research problems that are the key steps for achieving this goal. These include: 1) we investigate the online knowledge connection in software engineering by analyzing the knowledge network formed by Stack Overflow users' URL sharing activities. Through this study, we obtain an overall understanding of the domain knowledge organization, correlation and evolution, which inspires further research on extracting and linking software engineering knowledge. 2) we propose semi-supervised methods for extracting software-specific named entities, such as API mentions, from informal natural language text. 3) we develop automated techniques to link semantically linkable knowledge at document-level, and to link a recognized API mention to its fully qualified form as appeared in the API documentation at entity-level. We investigate the development and enhancement of NLP and IR techniques for the design challenges of these research problems brought by the socio-technical nature of software engineering social content. Extensive experiments show the effectiveness of our proposed approaches for analyzing and solving these problems.

Acknowledgements

The PhD journey is a process of self-quest and self-improvement. I am thankful to life who teaches me to be a better person.

I would like to thank my supervisors Dr. Zhenchang Xing, Dr. Nachiket Kapre, and Dr. Shang-Wei Lin. During the past several years, they have always been supporting and helping me on both technical research and the difficulties in life. Their encouragement and guidance lighted the way towards my PhD. I am very grateful for what I have learned from them.

I would like to thank Nan Zhang. Her optimistic character and wisdom in life have influenced me a lot. I want to express my gratitude to Yusen Li, who was always there to enlighten my spirit when I feel lost. I am thankful to my friends Robin Wu, Jacob Xu, Anthony Liew Kok Keong, Xuefeng Yao, Xiangwei Li, Tao Luo, Xiaozhe Gu, Sanjeev Kumar Das, Liwei Yang, and many others (apologize I could not mention them here one by one), who shared so much happiness with me during my PhD. I appreciate the efforts from my collaborators: Guibin Chen, Bowen Xu, Jing Li, Lingfeng Bao, Siddhartha, Dulitha Gunasekara, and an undergraduate student I supervised, Chee Yong Foo.

I have to express my deepest love to my parents and my relatives. Without their spiritual support, it would have been impossible for me to finish my PhD.

Finally, I would like to thank all the people who ever helped me.

Contents

Abstract	i
Acknowledgements	ii
Contents	iii
List of Figures	vii
List of Tables	viii
1 Introduction	1
1.1 Motivation	1
1.2 Research Overview	6
1.2.1 Understanding Domain Knowledge Connection	6
1.2.2 Extracting Domain Entities	7
1.2.3 Linking Domain Knowledge	9
1.3 Contributions	11
1.4 Thesis Outline	12
2 Related Work	13
2.1 Domain Knowledge Discovery	13
2.1.1 Knowledge Discovery in Stack Overflow	14
2.1.2 Analysis of Emergent Networks	15
2.1.3 Empirical Rules in Software Engineering	16
2.2 Extracting Software-specific Entities	16
2.2.1 Information Extraction in Software Engineering	17
2.2.2 Entity Extraction in Other Domains	20
2.3 Linking Domain Knowledge	21
2.3.1 Linking at Document Level	21
2.3.1.1 Link Prediction in Complex Networks	22
2.3.1.2 Linking Related Software Artifacts	23
2.3.2 Linking at Entity Level	24
2.3.2.1 API Linking	24
3 Understanding the Connection of Software Engineering Knowledge	26
3.1 Introduction	26

3.2	Research Methodology	28
3.2.1	Definition of Knowledge Network	28
3.2.2	Dataset	30
3.2.3	Overview of Research Questions	30
3.3	URL Sharing Activities in Stack Overflow — Basic Statistics	32
3.4	The Purposes of URL Sharing Activities — Understanding Why Users Share	35
3.4.1	Open Coding Process	35
3.4.2	Coding Schema	36
3.4.3	Coding Results	38
3.4.4	How Community Norms Shape the URL Sharing Behavior	38
3.5	Topological and Semantic Correlation — Analyzing Knowledge Interaction	39
3.5.1	Analyzing Degree Correlations	40
3.5.2	Analyzing Semantic Correlations	42
3.5.2.1	Directly Associated Knowledge Units	42
3.5.2.2	Knowledge Clusters	44
3.6	The Structure of Knowledge Network — Detecting Stability	47
3.7	The Dynamics of Knowledge Network — Revealing Evolution Trend	50
3.8	An Explanatory Generative Model	54
3.8.1	Baseline Probability — “To Reference or Not to Reference”	54
3.8.2	Information Value — “Create or Change the Distribution”	55
3.8.3	Preferential Attachment — “Reinforce the Distribution”	56
3.8.4	Simulating Knowledge Diffusion Process	57
3.9	Discussion	58
3.9.1	Implications for Q&A Site Design	58
3.9.2	Implications for Developers using Crowdsourced Knowledge	59
3.9.3	Implications for Software Engineering Knowledge Representation and Search	61
3.10	Summary	62
4	Extracting Software-specific Entities	64
4.1	Software-specific Named Entity Recognition	64
4.1.1	Introduction	64
4.1.2	Design Challenges	66
4.1.3	Problem Definition	68
4.1.4	The Software-specific NER System	71
4.1.4.1	Data Preparation	71
4.1.4.2	Customized Tokenization	74
4.1.4.3	Human Entity Annotation	75
4.1.4.4	Unsupervised Word Clustering	75
4.1.4.5	Supervised Learning based on CRF	76

4.1.5	Evaluation	78
4.1.5.1	Experimental Setup	78
4.1.5.2	Evaluation Metrics	79
4.1.5.3	Baseline System	79
4.1.5.4	Overall Comparison Results of All Entity Categories	82
4.1.5.5	Comparison Results of Individual Entity Category	83
4.1.5.6	Feature Ablation	84
4.1.5.7	Varying Labeled Data Size	85
4.1.6	Discussion	86
4.1.7	Summary	87
4.2	Recognizing API Mentions in Informal Text	88
4.2.1	Introduction	88
4.2.2	Approach Overview	92
4.2.2.1	Problem Formulation	92
4.2.2.2	Overview of Main Steps	93
4.2.3	Approach Details	94
4.2.3.1	Text Preprocessing	94
4.2.3.2	Constructing API Inventory	94
4.2.3.3	Training Sentences Selection and Labeling	96
4.2.3.4	Learning Word Representations	97
4.2.3.5	Iterative Self-Training Process	99
4.2.4	CRF-based Classifier	100
4.2.5	Experiment Setup	102
4.2.5.1	Tool Implementation	102
4.2.5.2	Studied libraries	102
4.2.5.3	Model Training	103
4.2.5.4	Human Labeling of Testing Data	104
4.2.5.5	Evaluation metrics	105
4.2.5.6	Baseline Methods for API Extraction	105
4.2.6	Experiment Results and Analysis	106
4.2.6.1	Overall Results for All API Mentions	106
4.2.6.2	Results for Each of the Studied Libraries	108
4.2.6.3	Feature Ablation	109
4.2.6.4	The Impact of Self-Training	110
4.2.7	Discussion	111
4.2.7.1	The Generality of Our Approach	111
4.2.7.2	Threats to Validity	112
4.2.8	Summary	113
5	Connecting Linkable Software Engineering Knowledge	114
5.1	Predicting Semantically Linkable Knowledge	
	– Connection of Knowledge at Document-level	114
5.1.1	Introduction	114
5.1.2	The Approach	118

5.1.2.1	Problem Formulation	118
5.1.2.2	Overview of Main Steps	118
5.1.2.3	Learning Word Representations	120
5.1.2.4	The CNN Architecture	121
5.1.2.5	Training the CNN	123
5.1.3	Experiment	124
5.1.3.1	Baseline Building	124
5.1.3.2	Dataset	126
5.1.3.3	Evaluation Metrics	127
5.1.3.4	Research Questions and Findings	128
5.1.4	Discussion	133
5.1.4.1	Qualitative Analysis	133
5.1.4.2	Error Analysis	134
5.1.4.3	Threats to Validity	136
5.1.5	Summary	136
5.2	Fine-grained API Linking	
	– Connection of Knowledge at Entity-level	137
5.2.1	Introduction	137
5.2.2	Background	139
5.2.2.1	Steps for API Linking	139
5.2.2.2	Ambiguities of API Mentions	140
5.2.3	Approach	141
5.2.3.1	Knowledge Base Population	142
5.2.3.2	Linking to Knowledge Base Entries	143
5.2.4	Evaluation	146
5.2.5	Summary	147
6	Conclusions and Future Work	149
6.1	Conclusions	149
6.2	Future Work	152
6.2.1	Mining Other Types of Software Artifacts	152
6.2.2	Towards Guided Search with Knowledge Location Map	153
6.2.3	Improving Fine-grained API Linking	155
6.2.4	Entity Relation Extraction in Software Engineering	156
A	List of Publications	158
	Bibliography	160

List of Figures

3.1	An Example of the Knowledge Network	29
3.2	Correlation Profile of Knowledge Units	41
3.3	Directly Associated Knowledge Units	44
3.4	Clustering Result of a Sub Knowledge Network Reachable from the Seed	45
3.5	Degree Distributions of Stack Overflow’s Knowledge Network	49
3.6	Indegree and Outdegree CCDF	50
3.7	JS Divergence between Consecutive States	53
3.8	JS Divergence between Each State and the Last State	54
3.9	Knowledge Diffusion Process through Social URL Sharing	56
3.10	A Stack Overflow Question Example	59
4.1	S-NER System Working Flow	71
4.2	Proportion of Different Categories of Software-specific Named Entities in Our Annotated Corpus	78
4.3	Comparison of S-NER and the Baseline System for the 10 Testing Sets .	82
4.4	Effects of Labeled Data Size on S-NER’s Performance	85
4.5	Illustrating Our Task	89
4.6	The Overview of Our Approach	92
4.7	The Impact of Self-training	111
5.1	Overview of the Main Steps	119
5.2	Example of Word Embedding	120
5.3	Architecture of the Proposed CNN	121
5.4	Illustrating the Task of API Linking	138
5.5	Approach Overview: API linking	142

List of Tables

3.1	Top 10 Referenced Domains	33
3.2	Purposes of URL Sharing	36
3.3	Topics of Knowledge Cluster 1	45
3.4	Top 1 Topic of the Six Detected Knowledge Clusters	46
4.1	Software-specific Entity Categories	69
4.2	Software-Entity Examples in Stack Overflow Posts	71
4.3	An Example of Our S-NER's Tokenization	74
4.4	Example Word Clustering Results	76
4.5	Overall Experimental Results	82
4.6	NER Results of Individual Entity Category	83
4.7	The Effects of Individual Feature(s)	86
4.8	A Subset of Variant Forms of API Mentions	90
4.9	Sentences Mentioning the Same API Form	91
4.10	Information of the Studied Libraries	102
4.11	Statistics of API Mentions in Testing Dataset	105
4.12	Comparison of Overall Performance	106
4.13	API Extraction Performance for Each of the 3 Studied Libraries	107
4.14	The Impact of One Kind of Feature(s)	110
5.1	An Example of Duplicate Knowledge Units from Stack Overflow	116
5.2	An Example of Directly Linked Knowledge Units from Stack Overflow	116
5.3	An Example of Indirectly Linked Knowledge Units from Stack Overflow (both have an link to 53513)	116
5.4	Accuracy Comparison	129
5.5	Precision, Recall and F1-Score of Our Approach and the Baseline Approaches	129
5.6	Performance of Our Approach with Word Embeddings Learned from Different Corpus	132
5.7	Examples of Different Classes of Semantically Linkable Knowledge Units	135
5.8	Data Source for Knowledge Base Population	142
5.9	Experimental Results of Our API Linking Approach	147

Chapter 1

Introduction

Our broad goal is to enable domain-specific knowledge graph and semantic search in software engineering to facilitate developer’s social information seeking process on the Web. In this thesis, we conduct a pioneering study identifying several important research problems on the domain knowledge organization and extraction aspects in software knowledge to move towards this goal. These include: 1) the understanding of domain knowledge organization and connection in software engineering, which inspires us to investigate further on: 2) software-specific entity extraction, in particular the extraction of API mentions; and 3) automated linking techniques to connect software engineering knowledge at both document-level and entity-level.

1.1 Motivation

Software engineering is a knowledge-intensive activity. Developers, whether novices or experts, frequently face different kinds of technical problems that cannot be solved using the knowledge they possess. In such situations, developers often seek help from their peers. With the advent of Web 2.0, social online communities, such as Stack Overflow and Quora, play a significant role in the social information seeking of software developers [1]. The user-generated content on these websites, referred to as software

engineering social content in this thesis, has grown into an important information resource on the Web that complements traditional technical documentations, and has been used extensively by developers for knowledge acquisition and sharing [2].

When reusing the textual content in these online communities, a fundamental information need of developers is to search for the discussions of a specific software entity (e.g., a library, a tool, an API, a development method), to find background knowledge, related concepts, usage patterns, bug solutions, or alternatives. For example, very often we see a developer search on the Web for the usage of an API of a certain library he is not familiar with, and the alternative APIs that can do a similar job to the APIs he already knows of. Such information need calls for innovative forms of information extraction, organization and search on software engineering social content. A very desirable goal would be to organize the information in a knowledge base about different software-specific entities and relationships between these entities. Such knowledge base can be represented as a graph, i.e., knowledge graph [3]. Search systems can exploit knowledge graph not only for finding the content that actually discusses a particular software-specific entity, but also to display additional facts and direct information about the central entity in a query [4]. Furthermore, we could also organize different pieces of software documents in the knowledge base based on their semantic relatedness, rather than manage each of them individually, so as to allow the return of more semantically meaningful content. Moving towards such domain-specific knowledge graphs and semantic search systems in software engineering is our long-term goal, which essentially requires us to organize software engineering knowledge centered around domain-specific named entities [5], e.g., APIs, and to build up the semantic linkages between different pieces of knowledge [3], e.g., linking semantically relevant discussions on the usage of an API. However, little research has been dedicated to these directions.

Existing research treats software engineering social content as textual documents, and uses information retrieval (IR) techniques, e.g., TF-IDF, to index the content. Such treatment on software engineering social content has 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 named entities [6–8]. Without extracting domain-specific entities from software engineering social content, it would be impossible to enable the far-reaching semantic search in software engineering. There have been some work in extracting API mentions (API is a typical type of software-specific entity) from developers’ textual discussions [9, 10], i.e., the importance of API mentions in text has been appreciated. However, these work heavily relies on observational orthographic heuristics to distinguish APIs from common words in a natural language sentence, which cannot be applied to API extraction in software engineering social content. Because in such social content, the simple name and informal synonyms of an API, rather than the fully or partially qualified API forms, are extensively used. Consequently, no distinct orthographic features are available.

On the other hand, automated connection of linkable knowledge is another important step towards knowledge graph and semantic search in software engineering: 1) the connection between documents could facilitate the knowledge diffusion process on the Web, and could also help the search engine return more relevant results. 2) the connection between a software entity to its formal source, known as entity linking in the literature, is essential to make sense of the unstructured textual data. However, existing work falls short to build up the semantic linkages among software engineering knowledge. First, at document-level, existing methods connect two software documents based on the textual relatedness between them, relying on human-engineered information retrieval based features [11–16]. These methods are unsuitable to detect the semantic relatedness for software engineering social content. Due to the crowdsourced nature, social developers may express the same piece of knowledge using different words and phrases and sentence structures, leading to few literal overlaps between two semantically relevant even equivalent knowledge. Also, existing work focuses on binary relatedness (i.e., related or not), while the relationships between different pieces of software engineering knowledge can be more fine-grained, as we will show later in Chapter 5. Second, at entity-level, the inherent ambiguous nature of API mentions in natural language text poses the opportunity for conducting domain-specific entity linking in software engineering. Existing work [10] links an API mention from text to its learning

resource, e.g., its official documentation. It uses the co-occurrence of surrounding APIs to help link the current API. However, it relies on simple regular expressions to detect the current API and contextual APIs, which is unsuitable for linking an API mention in software engineering social content.

Recent advances from the natural language processing (NLP) and information retrieval (IR) communities have already enabled semantics-oriented applications in general domains, such as knowledge graph and direct answers, to complement traditional keyword-based and document-centric search. These advances have greatly facilitated general users' information seeking on the Web. However, the software engineering community cannot benefit from these advances directly, because existing NLP and IR methods are not designed, thus not applicable, for analyzing software engineering social content which is both informal and technical. For example, existing named entity recognition (NER) methods from the NLP community recognize entities such as person, location and organization. However, these entities are rarely mentioned in software artifacts and are not what developers are concerned with. Also, the lexical writing forms and conversational context of software-specific entities (e.g., APIs) in software engineering social content are very different from those of a general-typed entity (e.g., location) mentioned in traditional text. What's more, the informal software engineering social content does not follow strict linguistic rules and is full of domain-specific terms, name variations, misspellings, synonyms, inconsistent annotations, which makes conventional NLP techniques perform poorly [17, 18]. Therefore, we must investigate the development and enhancement of IR and NLP techniques for the unique characteristics amplified by the emergence of software engineering social content.

In this thesis, we identify several key research steps along the path towards enabling domain-specific knowledge graph and semantic search in software engineering, using the developer-generated content on Stack Overflow as a case study. We choose Stack Overflow because it has become the most prominent online knowledge repository for computer programming [1, 2]. Our research ideas are briefed as follows.

Understanding knowledge connection. First and foremost, what is the connection status of software engineering knowledge, and how different pieces of knowledge are organized and correlated? It is important to obtain an overall understanding of the

knowledge organization in Stack Overflow, e.g., how different pieces of knowledge are connected and correlated to each other, and how the connection evolves. In Stack Overflow, due to the cross-reference of questions and answers, referred to as *internal URL sharing*, knowledge is diffused in the Q&A site, forming a large knowledge network. Such knowledge network is created manually through developers' day-to-day URL sharing activities. The network connection represents the relevance and relationships among software engineering knowledge. Mining users' purposes of URL sharing and the structure and dynamics of the knowledge network in Stack Overflow could help to understand of the status of domain knowledge connection and evolution, thereby inspiring further research on advanced knowledge representation and search.

Extracting domain-specific entities. As a foundational step towards knowledge graph and semantic search for the 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 known as named entity recognition (NER) in the literature. As mentioned, existing NER methods are not applicable for recognizing software-specific entities, such as programming languages, platforms, frameworks, tools, libraries, APIs and software standards. Therefore, we investigate the research problem of software-specific entity recognition. We are particularly interested in extracting APIs since correct API recognition is the basis for many software engineering tasks, such as API linking, bug fixing and API recommendation.

Connecting linkable domain knowledge. The current connection between different pieces of software engineering knowledge on Stack Overflow is opportunistic. The connection of related discussions on Stack Overflow is based on the community awareness of the presence of relevant questions and answers. And the connection between a software-specific entity (such as an API, or a library) and its official introductory Web pages (such as the official documentation introducing the API, or the Wikipedia entry to the library), is based on user-added manual links. Semantic linkages among domain knowledge, such as the textual semantic relatedness between software texts, and domain-specific entity linking, are essential to enabling knowledge graph and semantic search in software engineering. Therefore, we want to develop effective mechanisms to

semantically link different pieces of software engineering knowledge. We are also interested in studying domain-specific entity linking. Specifically, fine-grained API linking will be investigated, i.e., after we recognize an API mention from a Stack Overflow textual sentence, we link the recognized API to its formal form appeared in the official API documentation.

1.2 Research Overview

Inspired by these research thoughts mentioned in the previous section, we conduct several lines of work summarized as follows.

1.2.1 Understanding Domain Knowledge Connection

Consider a question and its answers and comments in Stack Overflow as a knowledge unit. Knowledge units are posted individually by different Stack Overflow users. As the volume of social content on Stack Overflow grows, there emerges different relationships between knowledge units, for example, knowledge unit A is similar to B, A is different from C, A can help solve the question of D, and so on. These relationships are an integral part of the crowdsourced developer knowledge.

At present, the underlying relationships among knowledge units are made explicit by Stack Overflow users' URL sharing activities, i.e., a user manually references URLs of relevant knowledge units when asking questions and providing answers. Through URL sharing, the knowledge in existing questions and answers is diffused to a new question or answer. As such, different pieces of knowledge units are connected. The behavior of URL sharing gives rise to an emergent knowledge network, i.e., a graph of knowledge units.

In Stack Overflow, why do developers share URLs? How is the community feedback to the knowledge being shared? What are the unique topological and semantic properties of the resulting knowledge network in Stack Overflow? Has this knowledge network become stable? If so, how does it reach to stability? Answering these questions

can help the software engineering community better understand the knowledge connection and knowledge diffusion process in programming-specific Q&A sites like Stack Overflow. We present a methodology to analyze URL sharing activities in Stack Overflow. We use open coding method to analyze why users share URLs in Stack Overflow, and develop a set of quantitative analysis methods to study the structural and dynamic properties of the emergent knowledge network in Stack Overflow. We also identify system designs, community norms, and social behavior theories that can help explain our empirical findings.

Our analysis allows us to make a number of interesting observations, including: Developers share URLs for various purposes, connecting different pieces of domain knowledge into a complex network of highly-recognized knowledge units; The knowledge network can be well partitioned into knowledge clusters that represent correlated topics; Although the size of the knowledge network in Stack Overflow keeps growing with the rapid growth of users and the fast development of software technologies, the knowledge units that developers care the most about are stable, which form the structural skeleton of the knowledge network; The knowledge network structure becomes stable quickly in Stack Overflow, but still, there are constant small structural changes with the burst of popularity of new software technologies or some newly recognized high-value knowledge units.

We further discuss the implications of our empirical study, which drive us to investigate further on domain-knowledge extraction and identifying semantic links between knowledge in software engineering.

1.2.2 Extracting Domain Entities

After we have an overall understanding of the knowledge connection in software engineering, we move on to another important step towards enabling semantics-oriented applications for software engineering. In Section 4.1 of Chapter 4, we develop a software-specific named entity recognition method to recognize the mentions of programming language, platform, API, tool-library-framework, and software standards from Stack

Overflow textual content. And in Section 4.2 of Chapter 4, we further propose an improved API recognition method.¹

Indeed, the importance of NER has been recognized by many research communities. It has been extensively studied on formal text (e.g., news articles [19]), informal text (e.g., emails [9, 20]), and social content (e.g., Tweets [21, 22]). Some NER work from other domain exists for recognizing domain entities, such as biomedical NER [23, 24], NER in clinical notes [25], and in commercial products [26]. The types of named entities need to be recognized are different in different domains. Accordingly, the NER method has to adapted based on the unique characters of the studied domain text. However, so far, there has been no dedicated work for NER in software engineering.

We perform software-specific NER in software engineering social content. The challenges lie in the unavailability of training data and the unique characters of Stack Overflow discussions which are both social and technical. Existing studies show that machine-learning based NER methods usually outperform rule-based methods [19, 21, 25]. However, building a machine learning based NER system requires a lot of annotated data for model training, while there exists no annotated data for NER in software engineering so far. Manually creating annotated data is tedious and prohibitively expensive. Moreover, Stack Overflow text is both informal and technical (software-specific). On the one hand, its informal nature makes conventional features such as part-of-speech (POS) and capitalization not reliable. On the other hand, many software-specific entity names are common words, and different software-specific entities often have the same name. To deal with these challenges, 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) [27] based learning, and a rich and effective set of features extracted from both labeled and unlabeled data for model training. 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,

¹Note that throughout this thesis, we use *entity extraction* and *named entity recognition* interchangeably, and we also use *API extraction* and *API recognition* interchangeably.

and to study the effectiveness of widely adopted NER techniques and features in the face of the unique characteristics of software engineering social content.

Through the evaluations of our S-NER method, we observe that API is the most concerned named entity type in software engineering, which constitutes more than 40% of the total number of entity mentions. However, S-NER’s performance for API recognition is much lower than that of other types of software entities, such as programming languages and software standards. The challenges for recognizing APIs from natural language text lie in the common-word polysemy and sentence-format variation issues, i.e., API names can be common English words when mentioned in text, and the contextual words of an API mention vary significantly from sentence to sentence. To handle these, we further 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. To make effective use of sentence context, we exploit state-of-the-art unsupervised language models, in particular class-based Brown Clustering [28, 29] and neural-network-based word embedding [30], [31] 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) [27], together with an iterative self-training mechanism (a.k.a. bootstrapping) [32], 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. 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 the API recognition of our S-NER method.

1.2.3 Linking Domain Knowledge

Different pieces of software engineering knowledge are correlated thus linkable.

At document level, we propose a deep-learning based approach for predicting multiclass semantically linkable knowledge unit on Stack Overflow. To better support more targeted information needs of developers, we consider four classes of semantic relatedness: duplicate, direct linkable, indirectly linkable, and isolated. To overcome the upper mentioned lexical gap issue, our approach recognizes and quantifies semantic relatedness between knowledge units using word embeddings [30], [33] and convolutional neural network (CNN) [34]. Our approach does not rely on human-engineered features to classify semantic relatedness. Instead, we collect large amounts of different types of user-created knowledge-unit links (duplicate, direct-linked, indirect-linked) from Stack Overflow to train the CNN to automatically learn the most informative word- and document-level features to classify the semantic relatedness between knowledge units. Our evaluation shows that our deep-learning based approach significantly and consistently outperforms existing methods using traditional word representations and human-engineered classifier features.

At entity level, the inherent ambiguities of API mentions in informal natural language text poses an opportunity for conducting domain-specific entity linking in software engineering, e.g., the simple name of a method can have multiple mappings in different classes of the same library or of different libraries. To resolve the ambiguities, we propose an API linking approach, leveraging mention-mention similarity, scope filtering, and mention-entry similarity, to determine which API entry in the knowledge base the recognized API mention actually refers to. In our case, for a recognized API mention, mention-mention similarity is implemented by checking if there exists build-in URLs for the same API mention and if fully qualified API names can be inferred from a similar API mention in the same knowledge unit. And mention-entry similarity is implemented by computing the cosine similarity between two TF-IDF vectors, which are converted from the content of the current discussion thread containing the mention and the content of API descriptions in the knowledge base, respectively. The scope filtering is mainly comprised of a set of Stack Overflow specific information, such as tags, question title and code blocks, to help eliminate the number of linking candidates. The evaluations show our approach can link API mentions in Stack Overflow discussions with high precision and recall.

1.3 Contributions

We expect this thesis to serve as a starting point towards enabling domain-specific knowledge graph and semantic search in software engineering to facilitate developer's social information seeking process. We make the following contributions.

- *Understanding domain knowledge organization in software engineering.* We perform the first systematic study towards understanding the connection of online software engineering knowledge by studying the structure and dynamics of the knowledge network formed by URL sharing in software-specific Q&A sites like Stack Overflow. We develop a semi-automatic methodology for qualitatively analyzing the purposes of knowledge sharing from the users' view, and for quantitatively analyzing the network connected by users' URL sharing. We expose the implications of our study on Q&A site design, knowledge accessibility, and further research on knowledge representation and search in software engineering.
- *Extracting domain-specific entities in software engineering.* We are the first to formulate the research problem of software-specific named entity recognition (NER). We identify design challenges and key design decisions to be made in NER for software engineering social content, based on which we develop a software-specific, machine-learning based NER method. Moreover, we develop a focused API recognition method that addresses a long-avoided challenge in API recognition, i.e., the ambiguity between the API sense and the normal sense of a common-word in informal natural language text. We tackle the challenge with demonstrated effectiveness by exploiting name synonyms and rich contextual features of an API mention in informal text.
- *Connecting linkable domain knowledge in software engineering.* We automate the connection between software engineering knowledge based on Stack Overflow in two aspects. First, internal to Stack Overflow, we predict semantically linkable knowledge units that connects knowledge at document-level. Specifically, we formulate the research problem of predicting multi-class linkable knowledge units in

Stack Overflow, which supports more targeted textual relatedness detection compared to existing work. We propose a deep learning based approach using convolutional neural network (CNN) to tackle the problem. Second, external to Stack Overflow, at entity-level, we propose a fine-grained API linking approach to link a recognized API mention to its formal form in the official API documentation, based on the results of our API recognition. Our API linking approach leverages mention-mention similarity, scope filtering, and mention-entry similarity to determine which API entry in the knowledge base the recognized API mention actually refers to.

1.4 Thesis Outline

This thesis is organized as follows. In Chapter 2, we review research aspects that are related to our work, including domain knowledge discovery in software engineering, information extraction in software engineering (particularly the extraction of software-specific keywords and APIs), and automated techniques for linking software artifacts. In Chapter 3, we aim to understand the connection of software engineering knowledge through mining the network formed by Stack Overflow users' URL sharing activities. Motivated by the implications brought by the empirical study in Chapter 3, we conduct software-specific entity extraction in Chapter 4, and we link software engineering knowledge in Chapter 5. Chapter 6 summarizes the contributions of this thesis, and discusses several possible extensions as future work.

Chapter 2

Related Work

In this chapter, we review existing research work that is related to this thesis. Our work mainly comprises of three aspects towards enabling semantic search in software engineering, including understanding knowledge connection, extracting domain-specific entity and connecting linkable knowledge in software engineering. Therefore, correspondingly, we categorize the related work into three categories as follows.

- In Section 2.1, we compare our empirical discoveries from analyzing the connection of Stack Overflow data with existing work on domain knowledge discovery.
- In Section 2.2, we summarize work on information extraction in software engineering, as well as some entity extraction work from other research communities.
- In Section 2.3, we report some representative research work on automatically identifying linkages between software artifacts at both document-level and entity-level, covering semantic relatedness between software textual content, and API linking (a.k.a., fine-grained traceability links recovery), respectively.

2.1 Domain Knowledge Discovery

Our work in Chapter 3 investigates the structure and dynamics of the knowledge network in Stack Overflow formed by users' URL sharing activities. Our research perspective, adopted methodology and empirical findings are related to several research

lines, including knowledge discovery in domain-specific Q&A sites like Stack Overflow, analysis of emergent networks, and empirical software engineering.

2.1.1 Knowledge Discovery in Stack Overflow

In addition to studies of general purpose Q&A sites like Yahoo!Answers [35, 36], domain-specific Q&A sites have also been extensively studied in recent years, among which Stack Overflow has become a programming knowledge base for software developers [2]. A community-curated list of publications using Stack Overflow data can be found at <http://meta.stackexchange.com/q/134495>.

Some researchers investigate topics and their trends in Stack Overflow [37–40] using topic models. For example, Barua et al. [38] study the general topics and their relationships and trends using LDA [41], while Rosen et al. [40] focus on what mobile developers on Stack Overflow talk about. Some analyze the contents and social connections in Stack Overflow to perform typical data mining tasks, such as prediction on user participation [42], expert identification [43]) and prediction [44], tag recommendation [45], recommending solutions to programming errors [46]), and how user activities correlate Q&A site properties [47]. Pal et al. [44] model users' selection preferences when answering questions to identify current experts and predict potential experts. Others study developers' behavior and social dynamics inside the cooperative community of Stack Overflow [1, 48–53]. For example, Tausczik et al. [54] analyze the impact of users' collaborative edits on the solution quality in MathOverflow for collaborative problem solving. Guerrouj et al. [50] examine the influence of App churn on Stack Overflow discussions of the corresponding App. Vasilescu et al. [48] investigate R programmers' transition from traditional mail lists to Stack Exchange. Combining statistical data analysis with user interviews, Squire [53] measures the utility of Stack Overflow for developer support. Mamykina et al. [1] find that productive competition, community credibility and designers' involvement with regular users are important factors for Stack Overflow's success. There is also research investigating sentiments in Stack Overflow discussions [55].

Compared with these existing work, our empirical study focuses on a unique perspective of Q&A behavior, i.e., URL sharing in Q&A sites. We reveal what kind of knowledge network URL sharing behavior brings to us and how Q&A site designers, software developers and software engineering researchers can benefit from an in-depth understanding of the formation and characteristics of the programming-specific knowledge network in Stack Overflow. The recent work by Gómez et al. [56] is the most similar work to ours. It also studies URL sharing in Stack Overflow, but it examines only some basic statistics of URL sharing activities, similar to the first two sub-questions (types of URLs shared and the size of the knowledge network) of our basic analysis in Section 3.3 in Chapter 3.

2.1.2 Analysis of Emergent Networks

The prosperity of social computing has incubated a number of emergent networks, such as user-user network in Q&A sites [57] and in Twitter [58], tag-tag network in social bookmarking systems [59] and in Q&A sites [45], and entity-entity network extracted from Wikipedia [60]. These networks enrich the general Web graph [61] by providing clean and dedicated knowledge sources.

Different from these networks, we study domain-specific knowledge network formed by ad-hoc Q&A activities in Stack Overflow. Our study reveals that the structural properties of Stack Overflow's knowledge network are very different from those of other types of networks. For example, the indegree distribution of Stack Overflow's knowledge network follows a strict power law [62], while its outdegree distribution does not. In contrast, both indegree and outdegree distributions of Twitter's user-user network follow a power law [58]. Broder et al. [61] show that both indegree and outdegree distributions of the general Web follow a power law, while Meusel et al. [63] analyze a more recent crawling data of the general Web and show that neither indegree distribution nor outdegree distribution follow a strict power law. The unique properties of Stack Overflow's knowledge network bring in the need for new knowledge search algorithms because network structure affects the performance of search algorithms as pointed out by Zhang et al. [57].

Researchers develop semi-analytical models to understand the stability of stochastic process. In social computing, these models have been widely applied to explain social tagging behavior and the generation of folksonomies in social bookmarking systems, such as the Polya Urn model adapted in [64], improved Yule-Simon model [65], semantic imitation model [66] and a generative tagging model [59]. In contrast, we propose an explanatory model based on information value [59] and preferential attachment [67] theories to explain how and why the structure of Stack Overflow’s knowledge network emerges from its users’ URL sharing behavior.

2.1.3 Empirical Rules in Software Engineering

Our finding regarding the stability of the programming-specific knowledge network in Stack Overflow, in terms of the knowledge units developers care the most about and the reinforcement of existing relationships among knowledge units, is an enrichment to the discovered power laws in the empirical software engineering context, for example, the “small world” effect in reverse engineering [68], 20% code having 80% errors [69], and power laws in modules comprising software systems [70].

2.2 Extracting Software-specific Entities

In Chapter 4, we perform software-specific named entity recognition (NER), and we also propose a method to improve the recognition of APIs (the most concerned software-specific named entity type) in informal natural language text. These work belongs to the category of software-specific information extraction. The methodologies we develop in these work are related to named entity recognition (NER) whose goal is to extract and categorize entities (e.g., location, people) in natural language text [21, 71, 72], and word sense disambiguation (WSD) whose goal is to disambiguate the sense of polysemous words in a given sentence context [73–75].

2.2.1 Information Extraction in Software Engineering

In the software engineering community, extracting information that developers care about from software artifacts has been extensively studied.

A large body of the work focus on API extraction either from natural language text or from source code [9, 10, 76–83]. In this line of research, the named entities being recognized involve and only involve APIs of certain programming languages. API extraction is an important software engineering task, not only the research on fine-grained API linking can benefit from it [9, 10, 76–78, 80, 84, 85], but many other software-related applications, such as API recommendation [86, 87] and bug fixing [88, 89]. Usually, in the software engineering community, API extraction is not studied alone and is studied as a pre-step for performing fine-grained API linking. While we introduce the state-of-the-art methods for extracting APIs from natural language text in this section, the related work to API linking will be covered in Section 2.3. Note that our work performs API extraction from informal natural language discussions among developers. Therefore, API extraction from code snippets using code parsers [77], e.g., Partial Program Analyzer (PPA), is related but is beyond the scope of our research focus.

Bacchelli et al. [9], [85] develop an API extraction and linking infrastructure, called Miler. They use lightweight regular expressions of distinct orthographic features to detect class and method mentions in developer emails. Six libraries of different programming languages, including Java, ActionScript, PHP and C, are studied. Significant performance differences across programming languages with and without distinct orthographic naming conventions are observed. For example, they show that the F1 score for Augeas (a C library whose APIs consist of a large number of lowercased common words) is only 0.24, which is much lower than the F1 score (0.46) for ArgoUML (a Java library whose APIs are camelcased, and are made of compound words). Their study shows the factors that significantly degrade the performance of lightweight regular expressions include: 1) common-word polysemy issue in API naming; and 2) non-standard API synonyms in the informal natural language discussions among developers, like emails. However, no working solution tackling the polysemy and synonym problems in API recognition is proposed in their work.

Dagenais and Robillard [10] 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 extract API mentions from natural language text using regular expressions similar to those of Miler [9]. That is, their API extraction from natural language text again relies on distinct orthographic features of API mentions. According to Bacchelli et al.’s empirical results [9], the performance of traceability recovery for APIs with no distinct orthographic features could be significantly degraded due to the low quality of API extraction. Incorrect API recognitions would propagate to the final API linking performance, which is particularly true when the recall of API recognition is low.

Island parsing is another popular technique for extracting information of interest from natural language text. By defining island grammars, the textual content is separated into constructs of interest (island) and the remainder (water) [90]. Bacchelli et al. [91] extract coarse-grained structured code fragments from natural language text with island parsing. However, the recognized code fragments are not further parsed to fine-grained APIs. Rigby and Robillard [76] 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 [76]. 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. In short, island parsing still requires orthographic features to define island grammars, thus it is unable to handle common-word polysemy issue and are sensitive to sentence-format variations.

Some other work recognizes important words or concepts in software artifacts to facilitate content comprehension [92–94]. For example, Capobianco et al. [92] employ part-of-speech (POS) tagging technique to differentiate noun terms from non-noun terms in software artifacts to improve the performance of traceability recovery. Shokripour et al. [93] first use part-of-speech information to find software noun terms in bug reports, and then use a simple weighting scheme of these noun terms to help assign bug reports to developers. However, in a piece of software text, not all these nouns identified by a POS tagger are of equal importance, e.g., some general concepts may

not be what developers are really concerned with. Furthermore, the POS taggers used in these work are general-purpose POS taggers that are not tailored for parsing software artifacts. As pointed out by Tian et al. [17] and Ye et al. [18], software engineering text, although expressed in natural language, is full of domain terms, software entities and software-specific informal languages. Therefore, the performance of general part-of-speech (POS) tagging technique degrades significantly on software engineering domain data, due to the lack of domain knowledge.

These above mentioned work falls short to extract fine-grained software entities from informal natural language text generated by developers, such as the social discussions on Stack Overflow, emails, and tweets from developers, because they require the entity mentions being written with distinct orthographic features. Furthermore, generally only one type of software-specific entity is considered, e.g., API. Comparing with these studies, we develop semi-supervised learning methods to recognize software-specific named entities mentioned in natural language text including APIs, framework-library-tool names, etc., which is more robust to the variations of the sentence context and word spellings.

As mentioned in Chapter 1, our research methodology focuses on the development and enhancement of IR and NLP techniques for the challenges amplified by the emergence of software engineering social content. From the perspective of domain adaption of IR and NLP methods in mining software artifacts, some related work includes: Witte et al. [95] build ontology representations for software artifacts. In [96], natural language parsing is used to classify the content of development emails. Bagheri and Ensan [97] mine Wikipedia contents and recommend Wikipedia entries as potential tags for tagging Stack Overflow posts. A recommended tag using their approach is naturally linked to relevant Wikipedia content. Sharma et al. [98] identify Tweets that contain software engineering knowledge using language model. Yang and Tan [99, 100] infer semantically related words appeared in source code written in Java and C. Similarly, Howard et al. [101] infer semantically similar words from Java code comments. Tian et al. [102], Yang and Tan [99, 100] and Howard et al. [101] study software-specific word similarity in software social texts.

2.2.2 Entity Extraction in Other Domains

Our work on extracting software-specific named entities (such as API mentions) from natural language text falls into the broad category of domain-specific named entity recognition (NER), a.k.a., entity extraction.

Traditional NER is performed on formal English texts such as news articles. One representative work of Traditional NER is done by Ratinov and Roth [19]. They systematically study the design challenges of NER in formal English texts (MUC7 dataset¹ and CoNLL03 dataset [5]), and a tiny set of 20 manually annotated webpages. Their key contribution is that they point out which design decisions should be made in traditional NER, including *what model to use for sequential inference*, *what features to choose*, *how to use external knowledge resources in NER*.

Apart from formal English texts, NER has also been widely studied in other genres of texts. We review some representative domain-specific NER work below.

NER in social media. The informal nature of social texts introduces new challenges to NER. Liu et al. [21] recognize entities from Tweets, and report an average F1 of 80% when recognizing persons, locations and organizations. Ritter et al. [22] also investigate NER in Tweets. They recognize a wider range of named entities, covering persons, locations, organizations, movie names, product names, company names, etc. They achieve an overall F1-score of 66%.

NER in bioinformatics. Biomedical-specific named entity recognition (Bio-NER) is an another active research field. Different from traditional NER that targets at person, location, organization, Bio-NER recognizes bio-specific entities such as protein, DNA, RNA and cell. Bio-NER has been raised as a community task [23]. Machine learning based systems are commonly used and are found to outperform rule based systems [24]. There are also research in other types of medical texts. For example, Wang introduces NER in clinical notes [25], which studies the recognition of organism names, medicine names, etc.

¹<https://catalog.ldc.upenn.edu/LDC2001T02>

Commercial products NER. NER also attracts great interests in the recognition of commercial product names. For example, Yao and Sun [26] perform mobile phone names recognition and normalization in Internet forums. Wu et al. [103] recognize mentions of consumer products from user-generated comments on the Web, such as the names of consumer electronics and automotive products.

NER in non-English natural language. The lexical and syntactic differences among different human languages drives language-specific NER research. For example, Wu et al. [104] propose a customized statistical model for NER in Chinese. Shaalan and Raza [105] develop a NER system for Arabic.

While we see the significance of NER in many domains, there has been no formal study of NER in software engineering. In Chapter 4 of this thesis, we formulate the research problem of NER in software engineering, and we propose a software-specific NER system. We discuss its domain-specific design challenges. And we reveal the key design decisions that further research should consider.

2.3 Linking Domain Knowledge

As mentioned, we explore methods to automatically connect linkable software engineering at both document-level and entity-level.

2.3.1 Linking at Document Level

Our work in Section 5.1 in Chapter 5 predicts semantically linkable knowledge in developers' discussions in Stack Overflow. It is related to two lines of research: link prediction in complex networks and semantic relatedness in software data.

2.3.1.1 Link Prediction in Complex Networks

Link Prediction focuses on detecting potentially linkable objects in an observed network that is complex and evolves dynamically. Link prediction in complex networks has attracted enormous attention from physics, biochemical, and computer science communities [106–112]. A significant proportion of the related research falls into link prediction in social networks, many of which have been found to be complex networks, e.g., the user-user network in online Q&A forums [57], in Twitter [58]. These social network research work aims to understand the network evolution patterns and identify potential social relations between users [107, 109, 112]. For example, Schifanella et al. perform social link prediction from shared user activities focusing on Flickr and Last.fm, two social media systems in which we can relate the tagging activity of the users with an explicit representation of their social network [112].

According to our empirical study in Chapter 3, the knowledge network formed by developers' URL sharing activities in Stack Overflow is also a complex network, i.e., the network structure is scale-free. We find that “the rich get richer” effect [67] influences the knowledge sharing activities and the growth of the knowledge network in Stack Overflow. As a result, a small portion of questions and answers attracts a large portion of developers' attention, while large amounts of relevant questions and answers in the long tail of the power law distribution are under linked. This finding motivates our work to predict potentially linkable knowledge units, which could enhance the knowledge sharing and search in Stack Overflow.

Unlike link prediction in social network which predicts only whether the two persons are linkable or not (i.e., binary classification), our work predicts different types of relatedness (duplicate, direct link, or indirect link) between potentially linkable knowledge units (i.e., multiclass classification). This multiclass link prediction is again inspired by our empirical study in Chapter 3, which reveals that users link relevant questions and answers for different purposes. As will be covered, our findings in Chapter 3 suggest that mixing different types of linkable knowledge units together could hinder the knowledge search as developers often need different types of relevant information in different situations.

2.3.1.2 Linking Related Software Artifacts

Measuring the relatedness (or similarity) between two pieces of textual contents has long been studied. The underlying mathematic model of textual contents has evolved from Vector-Space-Model (e.g., TF-IDF), n-gram language model, topic modeling (e.g., LDA [113]), to the recent development of neural language model (e.g., distributed word representations (or word embeddings) [30, 31, 33]). The trend is towards semantically richer similarity measures.

In the software engineering community, researchers utilize the textual similarity between two software artifacts to solve various software engineering tasks, such as duplicate or similar bug report detection [11–15], relevant software tweets detection [114], duplicate online question detection [16]. For example, Sun et al. use a Support Vector Machine (SVM) classifier in [15] and an IR based similarity measure (BM25F [115]) in [14], respectively, to detect duplicate bug reports. Nguyen et al. [12] combine the information retrieval based *BM25F* used in [14] with topic modeling (LDA) to further improve the detection accuracy of duplicate bug reports. Zhang et al. [16] detect duplicate Stack Overflow questions by comparing their titles and question descriptions using topic model (LDA). These existing work are formulated as a binary prediction problem, e.g., duplicate or non-duplicate.

In the NLP community, the development of convolutional neural network (CNN) architectures for sentence-level and document-level text processing is under intensive research. Some recent work utilizes CNN to learn the semantic relations between two pieces of texts. For example, Kim [116] proposes a simple CNN trained on top of Mikolov’s word embeddings [33], and then apply the CNN to sentence classification. Bogdanova et al. [117] apply CNN with word embeddings to duplicate question detection in online Q&A sites. These CNN-based methods outperform traditional NLP methods for sentence classification and duplication question detection.

Comparing with these existing works on semantic relatedness between two pieces of text, we leverage CNN with domain-specific word embeddings for the problem of multiclass classification of potentially linkable knowledge units in Stack Overflow, beyond the binary prediction of duplicate content or not.

2.3.2 Linking at Entity Level

2.3.2.1 API Linking

Research on API linking can be roughly divided into two categories based on the linking granularity, i.e., coarse-grained API linking [78, 80, 84] and the more recent fine-grained API linking [10, 76, 77]. Our work falls into the second category, i.e., fine-grained API linking.

Coarse-grained link refers to the link between two coarse-grained artifacts like an entire document and a source class. One important reason of creating such coarse-grained links is to ease software maintenance by bridging code blocks, e.g., class, to the corresponding section in the API documentation. For example, Antoniol et al. [78], Marcus et al. [80], Jiang et al. [84] perform coarse-grained API linking leveraging information retrieval techniques, i.e., vector space model (VSM), latent semantic indexing (LSI) and incremental LSI, respectively, to link the textual content of API documentation to source code of targeted systems. In these work, there are no fine-grained API recognition techniques involved, essentially the API mentions are preprocessed together with other texts and are represented as bag-of-words to be utilized by information retrieval methods.

By comparison, fine-grained API linking attempt to disambiguate which specific API the current API mention is referring to. Such fine-grained linking requires API recognition as the first step, as mentioned in Section 2.2. For example, in RecoDoc [10], Dagenais and Robillard first extract API mentions from text using regular expressions similar to [9], and then they devise a pipeline of filtering heuristics to resolve the ambiguities of API mentions. The most important filter in RecoDoc is their context filter, which checks the contextual API mentions of the current API mention. Such context filter has two limitations: 1) it does not use the other textual content other than APIs; 2) it heavily relies the correct recognition of API mentions. However, the API recognition technique is weak, i.e., only those API mentions with distinct orographic features can be captured by the regular expressions they develop. In [77], Subramanian et al. develop the Baker tool to extract API mentions from online code snippet on Stack

Overflow using partial program analysis (PPA), and then link the extracted APIs to their formal form appeared on the official documentation. The core idea of Baker is to utilize the co-occurrence of APIs in the same code snippet to iteratively reduce the number of linking candidates in the knowledge base. The artifact studied in Baker is code snippet, while we are concerned with API recognition and linking in online natural language text.

Essentially, considering an API as a named entity in the context of software engineering, API linking belongs to the general category of entity linking, a.k.a., entity disambiguation, which is the task of determining the identity of entities mentioned in text. A community-curated list of publications on entity linking can be found at <http://nlp.cs.rpi.edu/kbp/2014/elreading.html>. Entity linking requires a knowledge base containing the entities to which entity mentions can be linked. Wikipedia has been widely used as the knowledge base to be linked to in general domains. Contextual similarity based approaches are widely adopted [118–120]. We adapt the similarity based method used in [120] to our specific problem of API linking in Stack Overflow text, and we construct our own knowledge base consisting of information of APIs under different libraries.

Chapter 3

Understanding the Connection of Software Engineering Knowledge *

In this chapter, we aim to obtain an overall understanding of the knowledge connection in software engineering by studying the knowledge network formed by users' URL sharing activities in Stack Overflow.

3.1 Introduction

As mentioned, Stack Overflow arose as the media for social information seeking for the software engineering domain. When asking a question or providing an answer in Stack Overflow, users can reference URLs of existing questions and answers. We refer to this URL reference behavior as *URL sharing*. Through URL sharing, the knowledge in existing questions and answers is diffused to a new question or answer.

As introduced in Chapter 1, we consider a question together with its entire set of answers and comments in Stack Overflow as a *knowledge unit* regarding some programming-specific issues. The behavior of URL sharing gives rise to an emergent *knowledge network*, i.e., a graph of knowledge units. Studying why users share URLs to existing knowledge units, the community feedback of shared knowledge units, the

*The content of this chapter is based on our EMSE journal paper: “The Structure and Dynamics of Knowledge Network in Domain-specific Q&A Sites: A Case Study of Stack Overflow”.

structure of the knowledge network formed and how it is formed can be a way to know more in-depth the knowledge diffusion process in Stack Overflow, thereby enabling more effective knowledge sharing in the community, improving knowledge accessibility and navigability for software developers, and inspiring future research on advanced knowledge representation and search.

There have been several studies on hyperlink networks such as Wikipedia [121] and the general Web [61]. However, the knowledge network formed by URL sharing activities in domain-specific Q&A sites is a rich, and so far untapped source of information. Due to the domain-specific nature, the knowledge network formed by URL sharing in Stack Overflow is much smaller than other types of hyperlink network, such as Wikipedia and the general Web. Furthermore, the way that the knowledge network is formed in Stack Overflow is very different from that of other community-curated knowledge networks such as Wikipedia. Although both are formed through decentralized and collaborative efforts of a community of users, the structure of the Wikipedia knowledge network reflects the underlying structure of relevant knowledge. In contrast, individual knowledge units in Stack Overflow discuss ad-hoc questions posted by the developers. Knowledge units could be opportunistically referenced in other knowledge units based on the information value and relevance of the discussions and the community awareness of the presence of these discussions.

Considering the size of the knowledge network and the ad-hoc and opportunistic nature of URL sharing activities, would *stable* structure emerge in the knowledge network in Stack Overflow? By *stable*, we mean that the community has subconsciously developed implicit consensus about the relevance and relationships among knowledge units, and newly added relationships mostly reinforce already-present relationships. In this work, we propose a semi-automatic methodology (a mixture of qualitative and quantitative methods) for analyzing the knowledge network formed by URL sharing activities in Stack Overflow. Using the Stack Overflow data dump released on Mar. 16th, 2015, we aim to answer a series of research questions qualitatively and quantitatively: *why* Stack Overflow users share URLs, what *topological and semantic correlations* between knowledge units URL sharing activities create, whether the knowledge network formed

has become *stable* (i.e., *scale-free* in terms of degree distributions), *how* the knowledge network has evolved over time, and what *factors* and *theories* can help explain the structure and dynamics of the knowledge network in Stack Overflow.

Our research methodology and empirical results can be useful to many parties. Q&A site designers can use our methodology to study the knowledge diffusion process in other domain-specific Q&A sites. They can also use our results to determine how to support knowledge sharing more effectively in Q&A process. The knowledge network can be exploited to better support developers to search and navigate the knowledge repository in Stack Overflow. Further, our results can be extended to develop entity-centric knowledge representation, making searching semantic-aware and more intuitive for users.

3.2 Research Methodology

In this section, we define the knowledge network, describe the dataset we used and our research methods.

3.2.1 Definition of Knowledge Network

A *knowledge network* is a directed graph $KN(V, E)$. The node set V contains *knowledge units*. A knowledge unit consists of a question and its entire set of answers and other related information in the Q&A site. Formally, a knowledge unit is represented as a 3-tuple $\langle q, T, A \rangle$, where q is a question, T is the set of tags t_m ($1 \leq m \leq 5$ in Stack Overflow) of the question q , and A is the set of answers a_n ($n \geq 0$) to the question q . Each question or answer is referred to as a post in Stack Overflow. A post is represented as a 4-tuple $\langle id, body, comments, attributes \rangle$, where id is the post unique identifier, $body$ is the content of the post, $comments$ is the set of comments c_i ($i \geq 0$) associated with the post, $attributes$ is the set of metrics of the post (e.g., votes, viewcount, favorites).

The edge set E contains the *associations* from one knowledge unit to another. Let ku_s and ku_t be the two knowledge units. There is a directed association from ku_s to ku_t

FIGURE 3.1: An Example of the Knowledge Network

denoted as $ku_s \rightarrow ku_t$, if and only if the posts of ku_s reference at least one post of ku_t , i.e., the URL link of a post in ku_t is anchored or contained in the textual descriptions of ku_s . ku_s and ku_t are referred to as *source unit* and *target unit* respectively. We refer to the number of associations pointing to a knowledge unit ku as the *indegree* of ku , denoted as $indegree(ku)$, and the number of associations leaving from the ku as the *outdegree* of ku , denoted as $outdegree(ku)$. Indegree represents the degree of attention that a knowledge unit attracts in the community, and outdegree represents the amount of help that a knowledge unit seeks from others. If $indegree(ku) + outdegree(ku) = 0$, ku is an isolated knowledge unit, i.e., ku is not associated with any other knowledge units.

Figure 3.1 gives an example of knowledge unit, association, and knowledge network. The left-hand question (ID=4547310) and all its answers comprise knowledge unit 1 (ku_1), and the right-hand question (ID=53513) and all its answers comprise knowledge unit 2 (ku_2). ku_1 discusses whether there is an `isempty()` method for iterating a stack in Python. ku_2 discusses how to check if a list is empty in Python. Clearly, these two knowledge units are correlated. In fact, in Answer 2 of ku_1 , the answerer references the question of ku_2 , which creates the association $ku_1 \rightarrow ku_2$. ku_1 and ku_2 , together with their association, form a (smallest) knowledge network.

3.2.2 Dataset

For the analysis done in this chapter, the official Stack Overflow data dump that contains the data for the period of Jul. 31st, 2008 to Mar. 16th, 2015 has been used. To build the knowledge network as defined above, we scan the *posts* and *comments* tables of the data dump to extract knowledge units, and parse the *post_links* table to determine associations between knowledge units. Although the focus in this chapter is the internal URLs of Stack Overflow (i.e., the URLs with domain name *stackoverflow.com*), we also extract URLs external to *stackoverflow.com* from the body and comments of the posts. This allows us to compare the sharing activities of internal URLs of Stack Overflow with that of external URLs.

3.2.3 Overview of Research Questions

We hypothesize that Stack Overflow users can recognize the underlying relationships among knowledge units and make them explicit by referencing URLs of relevant knowledge units when asking questions and providing answers. We also hypothesize that, as time goes by, such URL sharing behavior will create a stable knowledge network that captures the important relationships among knowledge units. We validate our hypotheses through answering a series of research questions as raised in our Introduction Section. We also integrate our quantitative methods as a web service at <http://knowledge-so.appspot.com>.

RQ1. *What are the basic characteristics of URL sharing activities in Stack Overflow?*

In Section 3.3, we show that sharing of Stack Overflow internal URLs occurs much more frequently than sharing of external URLs. A large knowledge network with 2,378,327 non-isolated knowledge units and 2,271,053 associations has been formed in Stack Overflow. Importantly, the community feedback of the non-isolated knowledge units is statistically significantly higher than that of the isolated knowledge units.

RQ2. *Why do developers share URLs extensively in Stack Overflow?*

In Section 3.4, we use open coding method to identify, name and categorize the underlying purposes of URL sharing when developers ask questions and provide answers. We further explain our open coding results using Stack Overflow community norms.

RQ3. *How does developers' URL sharing behavior affect the semantic correlations of different knowledge units?* To be specific, do knowledge units correlate with other knowledge units with similar (or dissimilar) degrees or topics, which is known as assortative (or disassortative) mixing ¹?

In Section 3.5, we build the correlation profile [123] for the knowledge network. We demonstrate that knowledge units in Stack Overflow exhibit neither assortative nor disassortative mixing [122] in terms of degree correlations. We analyze the knowledge network reachable from a specific knowledge unit (results of the knowledge unit (question ID=53513) reported in this work). We show that the knowledge units exhibit high modularity ² and assortative mixing of semantic topics.

RQ4. *Is the knowledge network formed in Stack Overflow scale-free?*

In Section 3.6, we perform strict power law detection [62], rather than simple linear fitting, on the indegree and outdegree distributions of Stack Overflow's knowledge units, respectively. We show that the indegree distribution of knowledge units follows power law strictly, while outdegree does not (although visually it can be fitted using a straight line). This suggests that the indegree distribution of the knowledge network has become scale-free.

RQ5. *How does the indegree of the knowledge network evolve into its current structure?*

In Section 3.7, we use Jensen-Shannon divergence (JSD) [124] to study how a sequence of time-ordered degree distributions of knowledge units converge to the current distribution. After the convergence to the stable distribution, there can still be small but constant changes in the distribution over time.

¹In network science, assortative mixing is a bias in favor of connections between nodes with similar characteristics; disassortative mixing is a bias in favor of connections between dissimilarly characterized nodes [122].

²Modularity is one measure of network structure. A detected module is also called a community or a cluster. Networks with high modularity have dense connections between the nodes within modules but sparse connections between nodes in different modules.

RQ6. *What factors can help explain the structure and dynamics of Stack Overflow’s knowledge network?*

In Section 3.8, we propose an explanatory generative model based on information value [59] and preferential attachment [67] theories, to explain and simulate how the knowledge network has been formed in Stack Overflow.

Finally and importantly, we recall the implications of each section in this work and summarize and discuss these implications in detail in Section 3.9.

3.3 URL Sharing Activities in Stack Overflow

— Basic Statistics

First, we check if a valuable knowledge network can emerge from URL sharing activities in Stack Overflow.

What types of URLs have been shared? We identify 6,261,227 unique URLs from all the posts and comments in Stack Overflow. These URLs have been referenced 13,068,707 times in total. Table 3.1 lists the top 10 most frequently referenced domain names ordered by the Total column. The Total column shows the number of times that the unique URLs (shown in Unique column) from a domain have been referenced. The column % gives the uniqueness percentage, i.e. Unique/Total.

We can see that the domain *stackoverflow.com* stands out for both unique and total references, which means that developers reference Stack Overflow much more than any other domain. More than one sixth of the unique URLs reference to Stack Overflow’s own knowledge units. Furthermore, Stack Overflow internal URLs have higher uniqueness percentage than that of external URLs, except *jsfiddle.net*. *jsfiddle.net* is an online platform for rapid prototyping, testing, and sharing Javascript, CSS, and HTML code. Stack Overflow users often reference code examples shared on *jsfiddle.net* instead of embedding long code fragments in Stack Overflow posts. These code examples are usually unique for answering specific questions.

TABLE 3.1: Top 10 Referenced Domains

Domain	Unique	Total ↓	%
stackoverflow.com	1,303,641	2,179,791	59.8
microsoft.com	292,001	838,332	34.8
jsfiddle.net	691,280	757,489	91.3
github.com	371,401	679,592	54.7
wikipedia.org	70,549	397,721	17.7
google.com	175,870	374,030	47.0
php.net	50,221	288,377	17.4
oracle.com	71,681	271,933	26.4
android.com	25,029	198,683	12.6
jquery.com	9,052	185,464	4.9

What is the size of the knowledge network? As of Mar. 16th, 2015, Stack Overflow has 8,978,719 questions and 15,074,572 answers, i.e., 1 question has about 1.68 answers. The distribution of answers to questions is similar to what was observed in [49] (see Section 5 of their paper)¹. As such, the knowledge network we built from the data dump consists of 8,978,719 knowledge units, among which 6,600,392 are isolated knowledge units. The rest 2,378,327 non-isolated knowledge units have in total 2,271,053 associations. The number of non-isolated units and their associations reported here are calculated using the *post_links* table of Stack Overflow’s official data dump, which consists of all Stack Overflow’s internal references. The number of domain references shown in Table 3.1 is calculated by scanning Stack Overflow’s *posts* table and *comments* table, which consist of all Stack Overflow’s questions, answers, and comments. We have to point out that there exists minor inconsistencies among the officially released *posts* and *post_links* table. For example, table *post_links* says that post ID 2572 is linked to post ID 209329. However, both posts cannot be found from the *posts* table (they have been removed from Stack Overflow). As a result, the total number of associations (2,271,053) reported here is slightly different from the total number of references to *stackoverflow.com* reported in Table 3.1. For consistency, we will use the number 2,271,053 as the total number of associations in the remainder of this work.

In the early stage of Stack Overflow, its knowledge base is small and knowledge units are isolated. As Stack Overflow accumulates a sufficiently large amount of

¹We do not list the detailed answer-to-question distribution, since the focus of this work is on those non-isolated knowledge units and their associations.

knowledge units, similar or correlated knowledge units start appearing [125]. Such an accumulation leads to a time delay between the posting of a knowledge unit and its linking to other knowledge units. For instance, the first association in Stack Overflow was generated on Apr. 26th, 2010, about 24 months after the launch of Stack Overflow. We will discuss this further in Section 3.8.1 and Section 3.8.2.

How is the community feedback to non-isolated knowledge units versus isolated knowledge units? In Stack Overflow, users can archive *favorite* questions; an asker can *accept* an answer to his question as best answer; users can *upvote* questions and answers. In these cases, we consider the corresponding knowledge unit as having received positive feedback to the knowledge embodied in the knowledge units by the community.

To compare the community feedback to non-isolated knowledge units with that of isolated units, we control the number of answers of a knowledge unit to be no less than 5 to guarantee the popularity of the knowledge unit selected. We collect all the non-isolated knowledge units (group 1) and isolated knowledge units (group 2) with this level of popularity. For each knowledge unit in the two groups, we count the number of *votes* (sum of *upvotes-downvotes* of all its posts) and the number of users who *favorite* the question. If the knowledge unit has an accepted answer, we consider it as one *upvote* given by the asker. We measure the normalized community feedback to a knowledge unit as $votes + favorites$ divided by the number of posts in the knowledge unit.

We find that the median normalized community feedback to a non-isolated knowledge unit is 13.75, which is much higher than that of an isolated knowledge unit (1.48). We use median rather than mean as the metric, because the feedback metric has a right-skewed distribution. To build confidence with the hypothesis that non-isolated knowledge units are of higher positive feedback compared with isolated knowledge units, we perform Wilcoxon-Mann-Whitney test to the two groups of feedback. We get the p-value $p = 4.25e^{-210}$. Since $p \ll 0.05$, it means that the feedback difference between the two groups of knowledge units is statistically significant.

Implications. Our study indicates that the community feedback of non-isolated knowledge units is significantly better than that of the isolated knowledge units. This implies that non-isolated knowledge units have attracted much more user attentions when compared to isolated knowledge units. That is to say, the non-isolated knowledge units can potentially receive much more visits. The good community feedback of non-isolated units, together with their highly correlated nature which will be studied in Section 3.5, indicates the knowledge density of a non-isolated sub knowledge network is generally larger than an isolated one. This can potentially introduce navigation difficulties for novice developers, which will be further discussed in Section 5.1.4.

Overall, the basic analysis shows that URL sharing in Stack Overflow gives rise to a valuable knowledge network that warrants the further study.

3.4 The Purposes of URL Sharing Activities

— Understanding Why Users Share

Stack Overflow’s internal URLs are shared by developers for certain reasons. In this section, we infer the purposes of URL sharing from the content of the referenced posts and the discussion context where URLs are shared.

3.4.1 Open Coding Process

We randomly select 1,100 different URL references. 100 of these URL references are used to develop the coding schema, and then the rest 1,000 URL references are used to code the URL sharing purposes of Stack Overflow users. In our context, the overall population size is 2,271,053, which is the overall number of knowledge associations as reported in Section 3.3. Our sample size for coding is 1,000. Considering a confidence level of 95%, the margin of error caused by our sample size is as small as 3.1%. For each URL reference, we retrieve the source knowledge unit where the URL is referenced and the target knowledge unit that the URL links to. We then read the posts and comments of the two knowledge units to identify, name and categorize why the user shares a

TABLE 3.2: Purposes of URL Sharing

Category	Frequency
1.Reference information for problem solving	31.4%
2.Reference existing answers	20.5% (2a: 13.9%, 2b: 6.6%)
3.Visited but not helpful	19.4%
4.Recommend related information	25.6%
5.Others	3.1%

particular URL in the source knowledge unit. This open coding process involves 3 stages and is performed by 3 persons (P1, P2, P3) who are all from computer science background with 5+ years of programming experience.

In Stage 1, P1 first develop a draft coding schema (i.e., categories) of purposes of URL sharing using 100 randomly selected URL references. Then P2 and P3 use the draft coding schema to categorize the same 100 URL references collaboratively, during which the draft schema is revised and refined. At the end of Stage 1, we obtain 5 categories of purposes of URL sharing. In Stage 2, P1 and P2 apply the resulting coding schema of Stage 1 to categorize another 1,000 URL references independently. They are instructed to take notes regarding the deficiency and ambiguity of the coding schema for categorizing certain URL references. The inter-rater agreement (Cohen's kappa) of this stage is 0.82. In Stage 3, P1, P2 and P3 discuss the coding results obtained in Stage 2 to revolve the disagreements between P1 and P2, and to revise the coding schema to resolve schema deficiencies and ambiguities. At the end of Stage 3, we obtain the final coding schema and the final coding results of 1,100 URL references.

3.4.2 Coding Schema

The final coding schema contains five general categories of purposes of URL sharing in Stack Overflow as follows:

1. Reference information that can help to solve the problem.

There are four sub-categories: (a) The referenced post explains certain concept, approach, background knowledge, or the URL sharer's claim. (b) The referenced

post gives working examples, e.g., code snippets. (c) The referenced post covers a sub-step for solving a complex problem. (d) The answer is adapted from the referenced post.

2. Reference existing answers to a question

There are two sub-categories: (a) Confirmed duplicate. The question of the source knowledge unit is explicitly marked as duplicate by users with high reputation. Such questions can be easily recognized by the [duplicate] marker at the end of the question title. See an example at <http://stackoverflow.com/q/1725517>. (b) Possible duplicate. The URL sharer suggests that the question is duplicate to the referenced posts. However, the question asker and other users disagree with the suggestion. Possible duplicates can be judged by the comments or edit history of the question. See an example at <http://stackoverflow.com/q/23856165>.

3. Reference visited posts that cannot solve the problem.

Before asking a question, a user may “search and research” existing posts. If he does not find answers, he may reference those visited posts to let the community know what he had tried but failed. Such URL references are usually contained in the body or the comments of the question. See an example at <http://stackoverflow.com/q/23935102>.

4. Recommend related information.

The referenced post provides related information, although it does not directly answer the question.

5. Others.

This category contains URL references that do not fit into the above 4 categories. For example, when a user asks a question with very poor descriptions, another user may comment that the question is not clear and reference a good example that demonstrate how to ask question properly. See the reference embedded in the comments to this question <http://stackoverflow.com/q/22693431>.

While some of the above categories are distinctive (e.g., category **3** and category **2a**), we acknowledge that there are literal overlaps among some other categories. For example, a URL referencing a related concept can be category **1a** or category **4**. We regulate that if a URL reference is surrounded with problem-solving descriptions (e.g., <http://stackoverflow.com/a/13799882/2728388>), we categorize the URL reference as category **1a**. If the URL is suggested as “related” or “similar”, but the URL sharer provides no or little explanation (e.g., the URL in the comment of this post <http://stackoverflow.com/q/28902097/2728388>), we categorize the URL reference as category **4**.

3.4.3 Coding Results

Table 3.2 summarizes our final coding results. As we can see, Stack Overflow users share URLs mainly to reference information that can help to solve a complex problem (category **1**) and to recommend related information that may be potentially helpful (category **4**). The other significant reason for URL sharing is to inform the community existing posts that one had tried but found not useful (category **3**). Although category **2** (reference existing answers) accounts for 20.4% of URL references, the real confirmed duplicates (category **2a**) are only 13.9%, while the rest 6.6% are possible duplicates.

Threats to validity. First, it is almost impossible to develop a perfect coding schema with no overlaps among the categories. Second, we only label 1,100 randomly-sampled URL references. Further studies are required to confirm and generalize our coding schema. However, we believe our 3-stage open coding process and the small margin of error caused by our sample size should produce reliable coding results.

3.4.4 How Community Norms Shape the URL Sharing Behavior

Stack Overflow has a continuous community effort to identify and mark duplicate questions. Therefore, certain percentage of URLs links to existing questions that already have answers to the duplicate questions (category **2a**, 13.9%). However, as Stack Overflow encourages users “search and research” existing knowledge base

before posting a new question ¹, the percentage of confirmed duplicate questions is not very high. Furthermore, Stack Overflow encourages users sharing their “search and research” outcomes in their questions [125] so that the community knows what have been tried. This practice results in 19.4% (category 3) of URL references in our sampled dataset. Finally, Stack Overflow accepts only programming-related questions and has a strict policy on spam URLs ². As such, users have to be responsible for sharing relevant URLs. This explains why we do not find any junk URLs in our sampled dataset.

Implications. URL sharing connects different pieces of domain knowledge for various purposes. It also creates social contacts among users who hold related knowledge. Understanding the purposes of URL sharing and associating these purposes with user behaviors that Stack Overflow encourages inspire new search algorithms and bookmarking mechanisms to facilitate knowledge management and sharing in Q&A activities, which will be discussed further in Section 5.1.4.

3.5 Topological and Semantic Correlation

— Analyzing Knowledge Interaction

Analyzing the purposes of URL sharing suggests that referencing or recommending related or duplicate information (i.e., category 1, category 2, and category 4 in Table 3.2) is the main reason for URL sharing. How will these purposes affect the topological correlations (i.e., degrees of knowledge units) or semantical correlations (i.e., semantics of the contents of knowledge units) among associated knowledge units? Will URL sharing behavior results in assortative (or disassortative) mixing [122] of knowledge units in terms of their degree similarity (or dissimilarity) and semantic similarity (or dissimilarity)?

¹How do I ask a good question? <http://stackoverflow.com/help/how-to-ask>

²How does Stack Overflow handle spam? <http://meta.stackexchange.com/q/2765>

3.5.1 Analyzing Degree Correlations

Background. Given an association $ku_s \rightarrow ku_t$, the source knowledge unit ku_s seeks help and the target knowledge unit ku_t provides help. Generally, it holds that the most referenced knowledge units are most helpful for developers. Thus, if we regard the knowledge unit as a knowledge “helper”, the indegree of the knowledge unit reflects the helper’s “expertise” level. Degree correlations reveal the similarity (or dissimilarity) between the “expertise” levels of source and target knowledge units.

Method. To study the degree correlations of knowledge units, we construct the correlation profile [123] of knowledge units. A correlation profile summarizes the relative frequency of the combination of expertise-levels between help seekers ku_s and help providers ku_t in the knowledge network. Figure 3.2 shows the resulting correlation profile in contour plot. The x axis and y axis represent the expertise-level of source and target knowledge units respectively, based on their indegree values. The z axis represents the relative frequency of the combination of expertise-levels between “help seeker” ku_s and “help provider” ku_t , and its value is quantified in the color bar on the right. The larger the z value, the more frequently the corresponding combination of expertise-levels appears.

We perform logarithmical data binning to indegree values and frequencies of expertise-level combinations. In our knowledge network, the smallest indegree value is 0 (i.e., knowledge units with only outgoing associations). Thus, we add 1 to all indegree values to be able to calculate logarithm. We obtain 10 expertise-levels for knowledge units using natural log transformation. The smallest indegree maps to expertise-level 0, and the largest indegree 6447 becomes 8.77, mapping to expertise-level 9. We accumulate the number of knowledge units falling into each expertise-level, and compute the relative frequencies of expertise-level combinations. The relative frequencies expertise-level combinations map to 13 levels after logarithmical data binning.

Results. We can understand Figure 3.2 from three perspectives. First, looking at the overall distribution of expertise-level combinations, we can see that the dominant

FIGURE 3.2: Correlation Profile of Knowledge Units

combinations (the yellow and orange region) are the knowledge units with small indegree values, i.e., “novices”. Novices seek a large amount of help but also provide much help. They contribute the most to the knowledge diffusion. On the other hand, there are very few associations among knowledge units with large indegree values (the dark blue region), i.e., “experts”.

Second, looking at the *y-axis* which represent source knowledge units referencing others, we can see that source units with expertise-level 0-1 mainly reference to target units with low and middle expertise-level 0-5, sometimes reference target units with high expertise-level 6-9. Source units with middle expertise-level 2-5 sometimes reference to target units with expertise-level 0-7. Source units with high expertise-level 6-9 rarely reference other knowledge units. This follows our intuition that “experts” normally provide help, not seek help.

Third, looking at the *x-axis* which represents target knowledge units being referenced by others, similar expertise correlation analysis can be applied. Overall, we can see that with the increase of the expertise-level of the target knowledge units, the frequency of references drops drastically.

Implications. Knowledge units show neither assortative mixing nor disassortative mixing in terms of degree similarity or dissimilarity. That is, degree similarity or dissimilarity is not a decisive factor in URL sharing. This structural property, together

with knowledge units' degree distribution properties discussed later, informs Q&A site designers that general search algorithm should be adapted to improve knowledge search in Stack Overflow, because network structure is one potential factor that affects the performance of search algorithms, as pointed out by Zhang et al. [57].

3.5.2 Analyzing Semantic Correlations

Background. We exemplify the analysis of semantic correlations using the sub knowledge network reachable from the “seed” knowledge unit (question ID=53513), i.e., we traverse the knowledge network starting from the seed and collect knowledge units that are directly and indirectly linked to the seed. This seed knowledge unit discusses *how to check if a list is empty in Python*¹. We aim to understand the following two questions:

- How semantically similar are the seed knowledge unit and the knowledge units directly associated with it?
- In the knowledge network reachable from the seed knowledge unit, will there be latent knowledge clusters, if so, how these knowledge clusters are connected and how semantically similar these knowledge clusters are?

3.5.2.1 Directly Associated Knowledge Units

Method. The body and comments of the posts in a knowledge unit are first transformed into a text document. We pre-process the text documents by filtering away stop words, discarding code snippets, and removing all HTML tags. The text documents are then transformed into tf-idf (term frequency-inverse document frequency) vectors [126]. The semantic similarity between the two knowledge units are measured by the cosine similarity of their tf-idf vectors.

Results. Figure 3.3 shows the seed knowledge unit (the central node) and all its 12 directly associated knowledge units. In this example, all the 12 knowledge

¹Visit the web service we built <http://knowledge-so.appspot.com> to see more examples.

units reference the seed knowledge unit. The central node, i.e., seed knowledge unit, discusses *how to check if a list is empty in Python*. The color of its directly associated nodes denotes the URL sharing purpose according to Table 3.2. The three words on top of the nodes are the top words given by tf-idf. The length of an edge is inversely proportional to the cosine similarity between the two knowledge units it connects. We label the uses of these 12 URL references according to coding schema in Table 3.2. The results are: 2 knowledge units reference the seed unit as it helps solve their problems (category 1). 4 knowledge units reference the seed unit as existing answers to their questions (category 2), including 3 knowledge units explicitly marked as [duplicate] (category 2a) and 1 knowledge unit suggested as possible duplicate (category 2b). 2 knowledge units recommend the seed knowledge unit as related information (category 4). 4 knowledge units list the seed knowledge units as visited but not helpful to their questions (category 3).

The length of an edge is inversely proportional to the semantic similarity between the two knowledge units it connects. We label each knowledge unit with the three words (excluding the word “python”) that have the highest tf-idf values, and label the edges with the cosine similarities between the two knowledge units. We can see that not all knowledge units are of equal semantic similarity to the seed knowledge unit. The smallest cosine similarity is 0.17, and the largest is 0.55. Some knowledge units discuss topics very similar to that of the seed knowledge unit, such as those labeled with words “list”, “empty”. Others discuss more remotely related topics such as “imap”, “var”, “exist”.

Implications. Although directly associated knowledge units exhibit assortative mixing of topics, degree of semantic similarities between associated knowledge units do not correlate with specific purposes of URL sharing. Surprisingly, knowledge units referencing the seed knowledge unit as “visited but not helpful” have relatively higher semantic similarities with the seed unit than many other knowledge units with other categories of purposes. These results call for innovations in non-keywords-based knowledge search systems that could more accurately assess the information value and relevance of the knowledge units, beyond simple term frequency similarity.

FIGURE 3.3: Directly Associated Knowledge Units

3.5.2.2 Knowledge Clusters

Method. We use breadth-first graph traversal to collect all the knowledge units reachable from the seed knowledge unit. The direction of the association is not considered when we perform the graph traversal. Surprisingly, the resulting sub knowledge network is of huge size, containing 1.04 million knowledge units (43.7% of all non-isolated knowledge units) and 1.32 million associations (57.8% of all associations). In order to unveil the underlying topological and semantic correlations in the knowledge network reachable from the seed, we limit the breadth-first traversal to obtain a sub knowledge network of 300 knowledge units and 369 associations.

To find out whether knowledge units form latent knowledge clusters, we perform community detection in the resulting sub knowledge network using the *Louvain Method* [127] implemented in Gephi [128]. The *Louvain Method* adopts modularity maximization iteratively to partition the graph into a finite number of disjoint clusters that are considered as communities. Then, we apply the Latent Dirichlet Allocation (LDA) [41] topic modeling to extract topics for the detected clusters of knowledge units. For each knowledge cluster, we detect 10 topics and run LDA for 1000 iterations.

Results. The community detection result is shown in Figure 3.4. The seed knowledge unit is marked as *S*, which is the same one as shown in Figure 3.3. The size of the nodes is proportional to their indegree. Nodes in the same community (i.e., knowledge cluster) share the same color. We have two interesting observations: (i) Knowledge units

FIGURE 3.4: Clustering Result of a Sub Knowledge Network Reachable from the Seed

TABLE 3.3: Topics of Knowledge Cluster 1

Topic ID	Dirichlet Parameter	Top Words
0	119.49245	list empty python check false boolean value true object array
1	12.8508	index answer reason execute len key blank type list python
2	9.77833	variable implicit collect python effect nonzero complex implement

in this resulting sub knowledge network can be very well partitioned into six communities. This indicates that Stack Overflow’s knowledge network exhibits high degree of modularity (modularity coefficient is 0.528 in this subnetwork). (ii) Knowledge clusters are connected to one another through either some hub nodes such as S, H2, H4 and H6, or some betweenness nodes such as B1, B3 and B5. The hub nodes are the knowledge units with high degree (indegree + outdegree), while the betweenness nodes represent the knowledge that are common in two or more knowledge clusters. These hub nodes and betweenness nodes form a structure skeleton for this sub knowledge network.

Table 3.3 summarizes the top 3 topics with the largest Dirichlet parameter for the knowledge cluster 1 that contains the knowledge units in Figure 3.3. We can see that the first topic dominates and should be considered as the main topic. The top words of the first topic, such as “python”, “list”, “empty”, “check” match particularly well with the semantics of the knowledge units shown in Figure 3.3. Table 3.4 lists the top 1 topic with the largest Dirichlet parameter of all the six knowledge clusters. We can

TABLE 3.4: Top 1 Topic of the Six Detected Knowledge Clusters

Cluster ID	Top Words
1	list empty python check false boolean value true object array
2	python check test method list catch key index string find
3	python attribute way check object method function exist type
4	type python function check method argument class code differ
5	valid check file window function way string directory exist
6	input user number integer valid python string error way check

see the topics of these six knowledge clusters are similar to certain extent, but are of apparent difference as well. Directly connected knowledge clusters exhibit assortative mixing. As the distance between knowledge clusters increases, the semantic similarity of knowledge clusters decreases.

To understand why this sub knowledge network exhibits these six knowledge clusters, we analyze users' URL sharing purposes among the seed unit, hub units, and betweenness units. Take the knowledge unit S, B1, H2 and H6 as an example. The seed unit S discusses *best way to check if a list is empty in Python*. The betweenness unit B1 (ID=11786157) discusses how to write Python code such that *in a nested if-statements, if certain list indexes exist, then run a function*. The knowledge unit (ID=23365216) between B1 and S can be easily solved by combining the answers in S and B1, and thus is marked by the community as [duplicate] of both S and B1.

In an answer of B1, one user mentions *try block* is the perfect candidate for handling the issue, which naturally introduces the knowledge of *exceptions* in Python. Another user suggests that to better understand Python *exceptions*, one has to know *EAFP programming principle* in Python. Thus, this user recommends a URL referencing the knowledge unit H2 (ID=11360858) that contains thorough knowledge on *EAFP principle* from its definition to usage. As such, the two knowledge clusters centered on S and H2 are bridged by the unit B1 that discusses issues relevant the knowledge of both clusters.

The knowledge cluster centered on H6 (ID=23294658) is the largest cluster in this sub-network. The question of H6 asks how to write Python code *asking the user for input until they give a valid response*. One user gives a comprehensive answer

(ID=2728388), which not only answers the question, but also covers a wide range of relevant Python knowledge such as *while-loop*, *exception handling*, and *user input checking*. The discussion on exception handling in this answer recommends the unit H2 for further information on *EAFP principle*, which connects the two clusters.

All the discussions in this answer are exemplified with working code snippets. As a result, this answer is highly recognized by the community and is regarded as a *community wiki level* answer. Thus, in many knowledge units regarding Python *while-loop*, *exception handling*, and *user input checking*, users recommend H6 as it likely has an answer to the questions. Interestingly, many knowledge units that were asked before H6 but received poor answers (e.g., question 10563920 and 24174288) are marked as [duplicate] of H6.

Implications. Navigating a large knowledge network can be challenging due to the reachability of the network and assortative mixing of semantic topics among associated knowledge units. High modularity of the knowledge network, semantic topics of knowledge clusters, and the structure skeleton formed by hub and betweenness knowledge units can be leveraged to improve the navigability of the knowledge network when knowledge users perform user-centered exploratory search [129] in the knowledge base.

3.6 The Structure of Knowledge Network

— Detecting Stability

We have seen how knowledge units correlate. In particular, we observed an emergent structure skeleton in the knowledge network formed in Stack Overflow. As this knowledge network is generated by a large community of users with little central coordination, we wonder if the structure of the knowledge network has become stable, in terms of the nodes' degree distributions, which motivates the study in this section.

Background. The structure of a complex network is stable if the network is

scale-free. A scale-free network should produce a power law degree distribution. A power law defines a functional relationship between two scalar quantities x and y :

$$y \propto x^{-\alpha} \quad (3.1)$$

where α is the scaling parameter. When performing $\log$ transformation, $\log(y)$ becomes a linear function of $\log(x)$. It is important to mention that real-world empirical distributions rarely fit into power law over the whole range of x values. Rather, observed power laws hold only over a limited range [130]. Hence, we have to determine a minimum value x_{min} so that the distribution of y follows a power law with $x \geq x_{min}$.

To detect a power law, a traditional method is to use linear fitting for the linear function between $\log(y)$ and $\log(x)$ [61]. Recent works [62, 63] suggest that simple linear fitting can lead to unreliable results. Thus, we adopt the strict power law detection method proposed in [62].

Method. We study indegree and outdegree distribution of Stack Overflow's knowledge network separately. Degree distribution is a function describing the number of knowledge units (i.e., y) in the network with a particular degree (i.e., x). We first transform the degree distribution of knowledge units into Complementary Cumulated Distribution Function (CCDF). For a degree value x , CCDF cumulates all the knowledge units whose degree is no less than x and divide the number of such knowledge units by the total number of knowledge units to get a probability, represented as $P(X \geq x)$. As suggested in [62], we use *maximum likelihood estimation* to estimate α , and we minimize *Kolmogorov-Smirnov (KS) statistic*, which quantifies a distance between the observed CCDF data and the theoretical power law model in Eq. 3.1, to find x_{min} .

We evaluate the power law fitting using statistical hypothesis testing. The null hypothesis is that the indegree (or outdegree) distribution follows power law. The alternative hypothesis is the indegree (or outdegree) distribution does not follow power law. First we generate a large number (10,000) of power law distributed synthetic data sets using the computed x_{min} and α . For each synthetic data set, we fit the data to its

FIGURE 3.5: Degree Distributions of Stack Overflow's Knowledge Network

theoretical power law model in Eq. 3.1 and calculate the corresponding *KS statistic*. Then we count the fraction of all the synthetic data sets whose resulting *KS statistics* are greater than the *KS statistics* of our empirical indegree (or outdegree) distribution data. This fraction is the *p-value* that quantifies the plausibility of the hypothesis. If *p-value* is no greater than the significance level, we reject the null hypothesis and accept the alternative hypothesis.

Results. Figure 3.5 gives indegree and outdegree distributions of the knowledge network. *x-axis* shows the degree value of the knowledge units, *y-axis* is the number of knowledge units with a particular degree. Visually both indegree and outdegree distributions seem to exhibit power laws when plotted in *log-log* space.

In Figure 3.6, we show the indegree (and outdegree) CCDF distribution (blue asterisks) and the power law fitting line (black line) obtained. For indegree, the power law fitting reports $\alpha=2.6$ and $x_{min}=26$. We see the power law fitting line coincides with the indegree CCDF distribution data very well. For outdegree, the power law fitting reports $\alpha=2.8$ and $x_{min}=1$. However, we see that the fitting line deviates from the outdegree CCDF distribution data.

The hypothesis testing shows that the *p-value* for indegree distribution is 0.71, while it is 0.00 for outdegree distribution. Therefore, we accept the null hypothesis for indegree distribution. That is, the indegree distribution of the knowledge network follows power law. In contrast, we reject the null hypothesis for outdegree distribution and accept the alternative hypothesis. That is, the outdegree distribution of the knowledge network does not follow power law.

FIGURE 3.6: Indegree and Outdegree CCDF

Implications. The structure of the knowledge network with respect to indegree distribution is scale free. That is, regardless of how large the network grows, the shape of indegree distribution remains largely the same and thus stable. This structural stability can be explained using preferential attachment theory [67], i.e., “the rich get richer”, as will be further covered in 3.8.3. For the knowledge community that Stack Overflow serves, this finding suggests that the knowledge units that developers care the most about are largely stable, although the size of the knowledge network keeps growing with the rapid growth of the users and the fast development of programming technologies. Furthermore, the indegree’s power law distribution implies that a small proportion of the knowledge units is attracting a large proportion of users’ attention. These knowledge units are usually hub and betweenness units that form the structure skeleton of the knowledge network. This structure skeleton can be leveraged to improve knowledge accessibility and navigability when knowledge users perform user-centered exploratory search [129] in the knowledge base.

3.7 The Dynamics of Knowledge Network

— Revealing Evolution Trend

In the previous section, we demonstrated that the indegree distribution of the knowledge network in Stack Overflow has reached stability. A stable knowledge network does not appear randomly and suddenly. Instead, it results from the distributive knowledge of the community over time. In this section, we continue to investigate how the structure of the knowledge network with respect to its indegree distribution has been gradually

formed from users' day-to-day URL sharing activities.

Background. The knowledge network in Stack Overflow has gone through an evolving process from the initial chaos to the current stability. We want to quantify the growth of the knowledge network over time. To that end, we divide its evolving process into a sequence of time-ordered *states* and measure the *differences* between these states.

In this work, we represent a state as the indegree distribution of rank-ordered knowledge units with the highest indegree values in the knowledge network at a time point. These top-ranked knowledge units represent hub and betweenness nodes that form the structure skeleton of the knowledge network. The *difference* between the two states is measured as the divergence between the two distributions. When the divergence becomes or is close to zero, it implies the emergence of structural stability in the knowledge network.

Method. We get a sequence of states at a sequence of sampled time points. Because the number of new associations created over a fixed period of time is uneven, we calculate the indegree distribution of ranked-ordered knowledge units in the knowledge network at flexible time points when a fixed number of new associations (10,000 in this work) has been created (similar to [130]), rather than computing distributions at fixed time intervals as in [59]. As a result, we get 227 states at 227 time points (i.e., 2,271,053 associations divided by 10,000).

At each sampled point, we select the top N ranked knowledge units with the highest indegree values, and transform these indegree values into a vector of N probabilities. The probability of the i th ranked knowledge units, denoted as $P(i)$, is computed as the indegree value of the knowledge unit at the i th ranked position divided by the sum of the indegree values of all the top N knowledge units. This probability vector represents the indegree distribution of a particular state.

We adopt the Jensen-Shannon divergence (JSD) to measure the divergence between the indegree distributions of the two states. JSD is based on the Kullback-Leibler divergence (KLD) but is symmetric and always computable [124]. Given two probability

vectors P and Q , with M as their average vector, i.e., $\frac{1}{2}(P + Q)$, the JSD between P and Q is defined as:

$$\text{JSD}(P \parallel Q) = \frac{1}{2}\text{KLD}(P \parallel M) + \frac{1}{2}\text{KLD}(Q \parallel M) \quad (3.2)$$

where

$$\text{KLD}(P \parallel Q) = \sum_i P(i) \log_2\left(\frac{P(i)}{Q(i)}\right) \quad (3.3)$$

Inspired by [59], we implement two complementary ways to examine how the indegree distribution of the knowledge network has evolved into a scale-free distribution.

- We calculate the JS divergence between the distributions of every two consecutive states.
- We calculate the JS divergence between the distributions at each state and the final distribution at the last state (i.e., Mar. 16th, 2015).

Results. Figure 3.7 shows JS divergence results between consecutive states. Here we show the results for the top 10, top 50, and top 100 knowledge units with the highest indegree values¹. For these three cases, we can see that they all start with high JSD values, which indicates that initial indegree distributions of the knowledge network are divergent between two consecutive states. After some changes in the beginning, the JSD values drop close to zero rapidly, before the total number of associations reaches 200,000, which is less than 10% of the total associations in the latest knowledge network. This happened about 12 months after the first association was created.

We zoom into the slopes before the JSD values converge (subfigure(a)). We find that the convergence speed for the top 10 knowledge units is the fastest, and the top 100 is the slowest, while the top 50 falls into the middle. This phenomenon is what we can expect. The more top knowledge units under investigation, the more time it requires for the community to develop some implicit consensus on which knowledge units are more worth referencing and sharing. Furthermore, even after the distribution

¹The results of other top N knowledge units are similar. Visit our web service at <http://knowledge-so.appspot.com> to see more results.

FIGURE 3.7: JS Divergence between Consecutive States

converges to a stable distribution, there can still be small changes in the distribution. The subfigure(b) in Figure 3.7 shows an example, which reflects the burst of references to some knowledge units with high information value (e.g., the question (ID=14220321) that discusses AJAX in JavaScript).

Figure 3.8 shows JS divergence results between the distribution at each state and the final distribution. We can see that the divergence between the initial distributions and the final distribution is much larger than the divergences between the two consecutive initial distributions. Furthermore, compared with the convergence of the consecutive distributions, the convergence to the final distribution takes much longer time, after about 1,300,000 associations have been created. This happened about 39 months after the first association was created.

Implications. The indegree distributions of rank-ordered knowledge units converge quickly. Then, small changes between consecutive states accumulate over time and finally make the indegree distribution stable. The convergence and changes of indegree distribution of the knowledge network can be explained using information value theory [59], as will be covered in Section 3.8.2.

FIGURE 3.8: JS Divergence between Each State and the Last State

3.8 An Explanatory Generative Model

From Section 3.3 to Section 3.7, we have progressively studied various aspects of Stack Overflow’s knowledge network, from its basic statistics, reasons of its formulation from the view of developers, how knowledge units correlate, whether stability has been reached, to how the knowledge network evolves.

In this section, to integrate previous findings, we propose an explanatory generative model that models the factors that affect developers’ URL sharing behavior in Stack Overflow. This model helps explain our empirical findings of the structure and dynamics of the knowledge network emerged from URL sharing behavior. Specifically, in Section 3.8.1, 3.8.2 and 3.8.3, we cover the three factors that affect the degree distribution of the knowledge network in Stack Overflow. In Section 3.8.4, we combine these factors into a model to simulate the process of Stack Overflow’s knowledge diffusion process.

3.8.1 Baseline Probability — “To Reference or Not to Reference”

As Stack Overflow accumulates a large amount of knowledge units, similar or correlated knowledge units start appearing [125]. As such, when asking a question or providing an answer in Stack Overflow, there will be a baseline probability P_r for a user to reference an existing knowledge unit. This baseline probability is determined by the priori knowledge of the users. A priori knowledge refers to what the users already know of and would like to share with others.

Users may obtain a priori knowledge by search, system recommendation, or bookmarking. Users can always “search and research” existing knowledge base before asking a question or providing an answer. If no desired knowledge units are found, the user can choose to ask his own Stack Overflow question. While an asker is entering the question title, Stack Overflow can dynamically prompt some existing questions that may already have answers to the new question that the user wants to ask. If the asker finds good answers in these existing questions, he does not even need to post the new question. Otherwise, he can reference those existing posts in his question to tell others what he already tried. Stack Overflow also allows users to bookmark a question as their “*Favorites*” if they find useful knowledge embodied in the question and its answers. They may reference to this bookmarked question in their answer if it can help with other questions.

3.8.2 Information Value — “Create or Change the Distribution”

Assume that a user decides to reference an existing knowledge unit. With a probability P_a , he references an existing knowledge unit based on the information value [59] of the knowledge unit. A knowledge unit will be referenced with a probability P_{iv} proportional to its information value. In the context of developers’ help-seeking process, the information value of a particular knowledge unit is judged by how much useful information is contained in the knowledge unit that can facilitate a developer to get the desired information he is searching for. The information value of a knowledge unit can be estimated using the attributes of the knowledge unit such as *votes*, *favorites*, and *best answer*, or using the semantics of the knowledge unit such as topics or programming-related entities and their relations.

The probability $P_a \times P_{iv}$ creates the initial degree distribution of knowledge units based on the information value of the knowledge units. At the early stage of Stack Overflow, all the knowledge units are either not referenced or only referenced a few times. At this stage, knowledge units with higher information value will have higher probability to be referenced. As such, a power law degree distribution of knowledge units could emerge after certain amount of URL sharing activities.

FIGURE 3.9: Knowledge Diffusion Process through Social URL Sharing

The probability $P_a \times P_{iv}$ can change the degree distribution by referencing the newly added high-value knowledge units and quickly move them up in the list of most referenced knowledge units. For example, the question (ID=14220321) was posted on Jan. 8th, 2013 and was first referenced on Jan. 10th 2013. As this question and its answers provide a good discussion about AJAX request in JavaScript, it was referenced frequently by other JavaScript-related questions and moved up to the top 10 most referenced knowledge unit on Nov. 26th, 2013. Such a burst of references to certain knowledge units due to their high information values leads to small changes in the degree distribution (see the subfigure(b) in Figure 3.7).

The probability $P_a \times P_{iv}$ can also affect the distribution by changing the relative ranking positions of knowledge units. For example, the knowledge unit (question ID=12573816) which discusses the C++ compiler's linker error ranked one position ahead of knowledge unit (question ID=218384) which discusses Java's null pointer exception before Jan. 31, 2015. However, during a short two-month period from Dec. 1st, 2014 to Jan. 31st, 2015, the knowledge unit (218384) had been referenced 365 times, which is 205 times more than that of the knowledge unit (12573816). This causes the swapping of the ranking positions between the two knowledge units.

3.8.3 Preferential Attachment — “Reinforce the Distribution”

The scale-free property of the indegree distribution of knowledge network, as shown in Section 3.6, indicates that new associations are more likely to be added to already

referenced knowledge units. Therefore, the indegree of those most referenced knowledge units increases proportionally to the number of new associations being added over time. This behavior is well known as preferential attachment [67], i.e., “the rich get richer”. When referencing an existing knowledge unit, with a probability $1 - P_a$, the user references one that has already been referenced. A highly referenced knowledge unit will appear in many posts. Thus, it is more likely to attract users’ attention and be shared again. Therefore, we model a probability P_{ind} that an already referenced knowledge unit will be referenced proportionally to its indegree. Let $R(x)$ be the times that a knowledge unit x has been referenced. Then, the probability P_{ind} is $\frac{R(x)}{\sum R(i)}$, where $\sum R(i)$ is the sum of the times that all knowledge units have been referenced. The probability $(1 - P_a) \times P_{ind}$ means that a knowledge unit with higher indegree in the past will more likely be reinforced in the future.

3.8.4 Simulating Knowledge Diffusion Process

Imagine the process of referencing a knowledge unit as putting a ball into bins as shown in Figure 3.9. Here a bin represents a knowledge unit, and a ball represents a reference from a knowledge unit to another (i.e., a knowledge association). A ball will be added to bins through one of the three paths. With the probability $1 - P_r$, the user does not put a ball in any bins (i.e., path 1), which means he does not reference any existing knowledge units. With the probability P_r (baseline probability), he puts a ball in one of the existing bins. Now, he has two choices. First, with a probability P_a , he puts a ball in a bin based on the information value P_{iv} of the bin (i.e., path 2). Second, with a probability $1 - P_a$, he puts a ball in a bin based on the number of balls P_{ind} already in the bin (i.e., path 3). By linearly interpolating the path 2 and path 3, we can generate the probability distribution of a knowledge unit x that will be referenced as:

$$P(x) = P_r(x) \times (P_a \times P_{iv} + (1 - P_a) \times \frac{R(x)}{\sum R(i)})$$

The information value dimension $P_a \times P_{iv}$ explains the creation and change of the power law distribution, while the $(1 - P_a) \times \frac{R(x)}{\sum R(i)}$ dimension explains the reinforcement and stability of the distribution.

3.9 Discussion

So far, we have obtained a comprehensive understanding of the structure and dynamics of the knowledge network in Stack Overflow. Finally, in this section, we discuss the implications of our study for Q&A site design, general users of crowdsourced knowledge base, and knowledge representation.

3.9.1 Implications for Q&A Site Design

A user must know of a knowledge unit before he can share it with others. To improve this baseline probability P_r of URL sharing in our generative model covered in Section 3.8.1, Q&A sites like Stack Overflow need effective mechanisms for users to search and bookmark existing knowledge units.

Q&A sites like Stack Overflow usually rank the search results by simply keyword matching, recency of latest action, or votes. These simple ranking criteria do not take into account the semantic topics of the knowledge units, nor the attention that the knowledge units attract in the community. As URL sharing behavior creates a knowledge network exhibiting assortative mixing of semantic topics and structural stability, a *Topic-Sensitive PageRank* algorithm [131] would provide more sensible rankings of the search results. Topic-sensitive PageRank could be applied to capture the importance of knowledge units in the knowledge network with respect to a particular topic. As Stack Overflow’s knowledge network exhibits different structural properties from the general Web, the original algorithm designed for the general Web must be adapted, as pointed out by Zhang et al. [57].

Stack Overflow allows users to bookmark knowledge units by *Favorites* (see the red box in Figure 3.10). In addition to such knowledge bookmarking, remembering the knowledge “searched and researched” just-in-time is also important for knowledge diffusion, as “search and research” is a common practice that Stack Overflow encourages, as mentioned in Section 3.4.4. Existing knowledge remembrance mechanisms like CiteHistory tool [125] lack effective knowledge representation to manage and retrieve knowledge at semantic level. The information is usually represented as a list of

FIGURE 3.10: A Stack Overflow Question Example

items (e.g., URLs), while what users care about are types, properties and relationships of domain-specific entities discussed in the knowledge units. A *knowledge graph* [132] would provide a semantic representation of bookmarked and just-in-time researched knowledge units (see also Section 3.9.3). The knowledge graph enables semantic search of the user's priori knowledge around domain-specific entities for knowledge sharing.

3.9.2 Implications for Developers using Crowdsourced Knowledge

Through years of accumulation, Stack Overflow has become a huge body of crowdsourced programming knowledge that complements traditional technical documentation [2]. This body of crowdsourced knowledge does not just help particular question askers or just Stack Overflow registered users. Instead, it can be accessed and used by anybody who has programming issues or wants to learn programming. For example, a person who wants to learn EAFP principle and exception handling in Python can visit the knowledge units H2 and H6 and other associated knowledge units shown in Figure 3.4 in Section 3.5.2.

Although the knowledge is highly likely out there, it may not be easily accessible in most situations. Developers, especially less experienced ones, who search the knowledge base in Stack Overflow are often faced with two difficulties: *navigability of knowledge network* and *search query formulation or reformulation*. When searching Stack Overflow, it is very likely that a user lands on a knowledge unit that is relevant but does not fully satisfy his information needs. To find the needed information, the user has to explore the knowledge units associated (directly or indirectly) with the current

knowledge unit or research with a new query. However, navigating Stack Overflow's knowledge network can be challenging due to the reachability of the network and different degrees of assortative mixing of semantic topics among associated knowledge units. Furthermore, for novice developers who hold insufficient knowledge for certain programming aspects, it would be difficult for them to precisely express the information needs.

When viewing a knowledge unit, Stack Overflow currently lists directly-associated knowledge units under "Linked" with question title and votes (see Figure 3.10). Looking at this list, the user has no idea about where the knowledge unit he is viewing stands in the larger context of the knowledge network, and have little information scent about where he can go, which knowledge unit(s) may lead to the knowledge he needs, or how he may reformulate his query to improve the search results. That is, knowledge accessibility becomes a critical issue for exploiting the crowdsourced knowledge in Stack Overflow. Furthermore, the difficulty in finding the needed information may also result in duplicate questions.

We believe high modularity of the knowledge network, semantic topics of knowledge clusters, and the structure skeleton formed by hub and betweenness knowledge units can be leveraged to support user-centered exploratory search [129]. The key idea, inspired by the observations in Section 3.5.2, is to make hidden knowledge clusters, their hub and betweenness unit, their topics and their correlations easily accessible in an *interactive knowledge location map*.

Figure 3.3, Figure 3.4 and Table 3.4 in Section 3.5.2 show some preliminary elements for building an interactive knowledge location map for large knowledge network. This knowledge location map is an analogy of location map that one can find in big shopping malls, in which shops (knowledge units) are "clustered" based on the goods (problems) they sell (solve), and some descriptions (topics) of each "cluster" are displayed aside. Location map also clearly tell shoppers (knowledge users) "You are here" in a big mall (knowledge network) and help them determine where they can go and how to get there. The extracted topics of available knowledge units or cluster, centered around programming-specific entities (also discussed in Section 3.9.3), could help with query reformulation. As such, interactive knowledge location map would improve

knowledge accessibility for users of the crowdsourced knowledge in Stack Overflow. Considering the fact that one quarter of the total knowledge units are connected (see Section 3.3), which is already a non-trivial proportion, plus the fact that more users' attention is there on the non-isolated knowledge units (see the implications at the end of Section 3.3), we believe the proposed knowledge location map is necessary for developers' exploratory search within Stack Overflow. Such a knowledge location map will be an additional channel that can provide extra assistance to developers' information seeking process without impeding the user's normal information search.

3.9.3 Implications for Software Engineering Knowledge Representation and Search

The knowledge network in Stack Overflow is built by the crowdsourced power of its users. It can be time consuming to purely rely on several human experts to digest and then integrate the large amount of information generated by millions of users into the knowledge network, as evidenced by the fact that there was a time delay between the posting of a knowledge unit and the referencing of the knowledge unit in other posts. This motivates advanced knowledge management in Stack Overflow that goes beyond human manual efforts. One such advanced technique is to develop entity-centric knowledge network/graph that organizes the knowledge in Stack Overflow centered on software-specific entities. A further motivation for developing entity-centric knowledge management systems is that existing search systems are mainly document-centric and topic-based; topics mainly capture linguistic properties of the knowledge units (i.e., documents), but they may not accurately represent the information value of the units, such as types, properties and relations of programming-related entities discussed in the units. As such, searching by topics often does not lead to knowledge units that meet the users' information needs. This is evident in those "visited but not helpful" URL references in Table 3.2, shown in Section 3.4.

Entity-centric search (or semantic search) [60, 133] is an open but far-reaching research area; it can better understand the users' information needs and determine the information value of the knowledge units. Unlike document-centric search that is built

on a hyperlink network of documents, entity-centric search is built on a knowledge graph that capture the types, properties and relationships of entities for a particular domain. Entity-centric search can enable more productive and pleasant user experience by providing direct, aggregated and in-depth answers around domain-specific entities that meet the user's information needs. Successful examples include Google's Direct Answer and Answer Panel and Facebook's Entity Graph.

The fast development and effective quality-control mechanism of Stack Overflow make it a well-accepted and trustworthy knowledge base for developers. As such, the knowledge network in Stack Overflow provides a solid basis for constructing a programming-specific knowledge graph that could more accurately capture the information value and relevance of the knowledge units. One potential work is to develop domain-specific named entity recognition (NER) [134] techniques to recognize different categories of programming-related entities embodied in the body and comments of knowledge units, such as programming languages, frameworks, tools, APIs, tutorials, and application features, which will be explored in the next chapter. A further direction is to develop domain-specific relation resolution [135] techniques to derive from the knowledge units semantic relations between the recognized programming-related entities, for example, an API implements an application feature, a code snippet fixes a bug of an API. The resulting programming-specific knowledge graph would enable semantic search of programming-related entities.

3.10 Summary

In this chapter, we presented a systematic study of the structure and dynamics of the knowledge network formed in Stack Overflow. This study shows: 1) Users share URLs for diverse categories of purposes. In other words, knowledge units can be linkable for various purposes. Being able to classifying different classes of linkable knowledge units would support more targeted information needs when users search or explore the linkable knowledge. For example, duplicate questions allow users to understand a problem from different aspects, directly linkable questions help explain concepts or sub-steps of a complex problem, while indirectly-linkable questions provide extended knowledge. 2)

These URL sharing behaviors create a complex knowledge network with high modularity, assortative mixing of semantic topics, and a structure skeleton consisting of highly recognized knowledge units. 3) The structure of the knowledge network with respect to indegree distribution is scale-free (i.e., stable), in spite of the ad-hoc and opportunistic nature of URL sharing activities, while the outdegree distribution of the knowledge network is not scale-free. 4) The indegree distributions of the knowledge network converge quickly, with small changes over time after the convergence to the stable distribution. We demonstrated that the knowledge network is a natural product of URL sharing behavior that Stack Overflow supports and encourages, and proposed an explanatory model based on information value and preferential attachment theories to explain the underlying factors that drive the formation and evolution of the knowledge network in Stack Overflow. In addition to our empirical study, we discussed the implications of our empirical findings for new search algorithm and bookmarking mechanism in Q&A site design, for improving knowledge accessibility in crowdsourced knowledge base, and importantly, for the far-reaching research on programming-specific knowledge graph and semantic search.

Based on the findings and implications of the study in this chapter, we investigate domain-specific entity extraction in Chapter 4, and we predict linkable knowledge at both document-level and entity-level in Chapter 5.

Chapter 4

Extracting Software-specific Entities

As mentioned, named entity recognition (NER) is a key step in enabling knowledge graph and semantic search. However, little work has been dedicated to the domain-specific NER in software engineering. In this chapter, we first formulate the research problem of NER in software engineering in Section 4.1. And we further propose a method to handle the polysemy and synonym issues in extraction API mentions (API is an important type of software-specific named entity) from informal natural language text in Section 4.2.

4.1 Software-specific Named Entity Recognition *

4.1.1 Introduction

As mentioned in Chapter 1, a fundamental task for reusing the social content on Stack Overflow 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. Mentions of software-specific entities in the content are treated in the same way as other regular textual content in existing approaches. Such treatment does not apply naively to recognize essential domain-specific entities [6–8].

*This section is based on our SANER-2016 conference paper: “Software-specific Named Entity Recognition in Software Engineering Social Content”.

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. With the broad goal of organizing online software engineering knowledge into a knowledge graph and enabling semantic search over it, we must first be able to perform NER to recognize mentions of software-specific named entities in software engineering social content and classify them into pre-defined categories. However, for software engineering texts, existing approaches are limited to dictionary look-up and rule-based methods based on code or text parsing techniques [76, 77, 83, 92]. Furthermore, the entity category is limited to only API.

In this section, 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: 1) What are the challenges in NER for software engineering social content, compared with formal text, other social content like Tweets, or other domain-specific texts like clinical notes? 2) How can we adapt state-of-the-art machine learning based NER pipeline for NER in software engineering social content? 3) 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 concrete contributions: 1) 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. 2) 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. 3) 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. 4) We provide our annotated corpus, collected gazetteers, unsupervised word clusters, and trained CRF models to the software engineering community for further research and validation.

4.1.2 Design Challenges

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. 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, 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 lower-CamelCase, 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) [27], 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 [136], shallow parsing [22], and NER [21, 25, 26].

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 [28]) learned from the abundant unlabeled data as extra features to improve accuracy of supervised NER model learned from small amount of annotated data [22, 31].

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 [19, 21, 137, 138]. 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.

4.1.3 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 $s = \langle w_1 w_2 \dots w_n \rangle$ ($n \geq 1$) 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,

TABLE 4.1: Software-specific Entity Categories

Entity Category	Annotation Tag	Examples
Programming Language	PL	<ul style="list-style-type: none"> • 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	<ul style="list-style-type: none"> • 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	<ul style="list-style-type: none"> • 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	<ul style="list-style-type: none"> • 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	<ul style="list-style-type: none"> • 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

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, we 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 4.1.

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.

In Table 4.2, 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 $w_1 w_2 \dots w_n$ 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 4.2: 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

4.1.4 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 4.1 shows an overview of our S-NER system. S-NER is based on Conditional Random Fields (CRF) [27] 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.

FIGURE 4.1: S-NER System Working Flow

4.1.4.1 Data Preparation

Labeled Data Preparation Supervised learning requires annotated (or labeled) data. Unlike previous studies [76, 77] 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 `<pre>` and `</pre>`. 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 [76, 77]; 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 `<code>` and `</code>` 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.

Unlabeled Data Preparation As mentioned in Section 4.1.2, 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 [28] 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.

External Knowledge Resources As mentioned in Section 4.1.2, 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 [19], 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 4.1 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 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

¹https://en.wikipedia.org/wiki/List_of_programming_languages

²https://en.wikipedia.org/wiki/Computing_platform

³https://en.wikipedia.org/wiki/List_of_operating_systems

⁴https://en.wikipedia.org/wiki/List_of_instruction_sets

⁵https://en.wikipedia.org/wiki/List_of_mobile_software_distribution_platforms

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 collect a list of data formats and technology acronyms from Stack Overflow tags.

TABLE 4.3: 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?

4.1.4.2 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 4.3, 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.

4.1.4.3 Human Entity Annotation

For annotation, we use Brat [139], 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 4.1, 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 to improve the consistency of our annotation.

4.1.4.4 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 [28, 29] on the unlabeled Stack Overflow posts (see Section 4.1.4.1). 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 Clustering¹. 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 4.4 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 4.4: 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
1111111111111	hover touch mouseover blur keyup keypress keydown mouseout fadeIn mouseenter mouseleave mousedown delegated show() fadeOut mouseup mousemove hide()

4.1.4.5 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.

Model Given a sequence of observations (tokens in this work) $\vec{x} = x_1, x_2, \dots, x_n$, we want to assign each observation with a label (from our annotation tags shown in Table 4.1 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 $\vec{y} = y_1, y_2, \dots, y_n$. In linear chain CRF, the probability vector of assigning $\vec{y}$ based on $\vec{x}$ is:

$$p_{\vec{x}}(\vec{y}|\vec{x}) \propto \exp\left(\sum_{j=1}^n \sum_{i=1}^m \lambda_i f_i(y_{j-1}, y_j, \vec{x}, j)\right) \quad (4.1)$$

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 .

¹<https://github.com/percyliang/brown-cluster>

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 4.1.4.4, 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 4.4 as an example. The length of this bitstring is 11. For prefix 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 ¹ for entries that consist of a span of words. If the fuzzy ratio is above 0.8, we consider it as a match.

4.1.5 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.

4.1.5.1 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.

FIGURE 4.2: Proportion of Different Categories of Software-specific Named Entities in Our Annotated Corpus

In Figure 4.2, we further show the proportions of different categories of software-specific entities according to our pre-defined entity categories in Table 4.1. 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

¹<https://github.com/seatgeek/fuzzywuzzy>

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 over-fitting, 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++¹, a popular CRF toolkit that has been widely used for sequential tagging tasks like NER.

4.1.5.2 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 4.2 in Section 4.1.3).

4.1.5.3 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 4.1.4.1). 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

¹<https://taku910.github.io/crfpp/>

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 ¹” is a JavaScript date library, “click ²” is a Python library, “Task ³” 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”

¹<http://momentjs.com/>

²<http://click.pocoo.org/5/>

³<http://taskphp.github.io/>

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”, “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 4.1.4.5).
- 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.

4.1.5.4 Overall Comparison Results of All Entity Categories

In Table 4.5, 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 4.1. 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 4.5: 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% ↑

FIGURE 4.3: Comparison of S-NER and the Baseline System for the 10 Testing Sets

Recall that we use 10-fold cross validation. In Figure 4.3, 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).

4.1.5.5 Comparison Results of Individual Entity Category

We report the experimental results of each individual entity category in Table 4.6. As we can see, the NER performances of both S-NER and the baseline system on different categories of entities are quite different.

TABLE 4.6: 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

4.6a. 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 4.1.4.5, and is more robust (93.358% F1).

In Table 4.6b, 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 4.6c, 4.6d, 4.6e, 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 4.5. 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.

4.1.5.6 Feature Ablation

We also perform independent feature ablation experiments to study the effect of individual feature(s) on the NER performance. In Table 4.7, 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.

FIGURE 4.4: Effects of Labeled Data Size on S-NER’s Performance

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 [19, 21], especially for NER in social texts such as Tweets [21]. In these studies, gazetteers can boost the NER performance tremendously by an F1 increase of 19% as reported in [21]. We will discuss more about the design of software-specific gazetteers in Section 4.1.6.

Our results show that initial capitalization feature is not particularly useful for software-specific NER, as we raised in Section 4.1.2. 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.

4.1.5.7 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 4.4. 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 4.7: The Effects of Individual Feature(s)

	F1-score for each entity category (%)					Overall F1(%)
	PL	Plat	API	Fram	Stan	
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

4.1.6 Discussion

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 4.7, 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 [19, 21, 137, 138].

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 Equation 4.1) 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 [137, 138], similar to what we do for the baseline system, as covered in Section 4.1.5.3. 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.

4.1.7 Summary

In this section, 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 for community validation and further research: <https://drive.google.com/open?id=0ByoLWPpAxGVFdEROT09EMC0zUzg>.

4.2 Recognizing API Mentions in Informal Text *

In the previous section, we proposed S-NER for recognizing general software-specific named entities, including programming languages, platforms, tool-library-frameworks, APIs, and software standards. We have shown that API is the most concerned entity type in developers' natural language discussions. However, S-NER's performance on recognizing APIs is much lower compared to the recognition of other types of software entities. S-NER does not perform well on handling the polysemy and synonym issues [85], [9] lying in the recognition of APIs from informal natural language text.

4.2.1 Introduction

In this section, we focus on API recognition. For example, Fig. 4.5 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.

Indeed, the importance of fine-grained API extraction from natural language sentences has long been recognized. Representative techniques include language-convention based regular expressions [85], [9], [10] and island parsing [91], [76]. 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

*This section is based on our ICSME-2016 conference paper: "Learning to Extract API Mentions from Informal Natural Language Discussions".

and consistently used, e.g., extracting *camelcased* Java APIs from *official* documentations [10]. However, they fall short to address the following two fundamental challenges in API extraction from *informal* natural language texts:

FIGURE 4.5: 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 `apply`, we can easily recognize them. Unfortunately, this is not always the case due to the informal nature of the texts. Then, such common-word APIs appear just as common words in a sentence, as shown in Fig. 4.5. In fact, the token *apply* (using the software-specific tokenizer we developed in Section 4.1.4.2 in Chapter 4) 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 greater than 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. 4.5. For API extraction from developers’ informal discussions, e.g., emails, Bacchelli et al. [9] 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 API mentions with distinct orthographic features in existing work [76]. There has been no work from the software engineering community addressing the polysemy problem in fine-grained API extraction.

TABLE 4.8: 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

- *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 4.8 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 4.9, 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 4.9, to

TABLE 4.9: 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
13741439	Apply can also do aggregation and other things
13761574	You are looking for <code>apply</code> (merge is like a database join.)
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.

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 [71], [21] 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).

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 [28, 29] and neural-network-based word embedding [30], [31] 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 [27], together with an iterative self-training mechanism (a.k.a. bootstrapping) [32], 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

FIGURE 4.6: The Overview of Our Approach

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 [9], island parsing [76], and our S-NER method covered in the previous section in this chapter. Our approach consistently and significantly outperforms the three baseline methods.

4.2.2 Approach Overview

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

4.2.2.1 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. 4.5. Specifically, 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 we are tackling common-word polysemy and sentence-format variations issues in recognizing whether a token is an API mention of certain libraries. Therefore, the recognition of APIs that are written in the polysemous common-word form is the focus of this work.

4.2.2.2 Overview of Main Steps

Figure 4.6 shows the main steps of our approach. In this study, we obtain natural language sentences from Stack Overflow discussions (the data used in this section is from Stack Overflow January 2016 data dump), to train and evaluate our approach (Section 4.2.3.1).

In our approach, we use two unsupervised language models (i.e., class-based Brown clustering [28] and neural-network based word embedding [31], [30]) to learn word representations from unlabeled text and cluster semantically similar words (Section 4.2.3.4). 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 4.2.3.2). 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 4.2.3.4). Third, unsupervised word clusters, together with self-training process (Section 4.2.3.5), 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 4.2.4). 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 4.2.3.5), through which high-confidence machine labeled sentences, together with human labeled sentences, are used to retrain the CRF model.

4.2.3 Approach Details

We now describe each component of our approach in detail.

4.2.3.1 Text Preprocessing

The text preprocessing is similar to what has been covered in Section 4.1.4.1, including code snippet removal, HTML tags cleaning, and software-specific tokenization.

4.2.3.2 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, from software-specific NER work in Section 4.1, we see that a gazetteer of standard API

names contributes only marginally to the NER performance. This is because of the wide presence of non-standard API synonyms and their derivational forms in informal natural language text (see Table 4.8 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 [21, 71] and WSD [73, 74] 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 4.2.3.4) 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 4.2.4); 2) ensure that training data and test data reaches a good coverage of polysemous and derivational forms of library APIs (see Section 4.2.3.3).

4.2.3.3 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 [73, 75].

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 [71] and the mobile phone name extraction work [26], 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 4.2.3.5.

4.2.3.4 Learning Word Representations

In informal social discussions, both API mentions and sentence context vary greatly (see Table 4.8 and Table 4.9). These variations result in out-of-vocabulary issue [71] 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 state-of-the-art unsupervised language models include class-based Brown clustering [28, 29] and neural-network based word embedding [30]. Different language models make different assumptions about corpus properties to evaluate the semantic

similarity between words. Empirical studies [140] and [141] 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 4.2.3.1. This produces a large set of unlabeled sentences. Unlike prior work [140], [141], [71], [26], 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 [28] outputs a collection of word clusters. Each word belongs to one cluster. Words in the same cluster share the same bitstring representation. Studies [26], [71] 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 4.2.3.2).

For neural-network based word embedding, we use continuous skip-gram model [30, 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 [30, 31]. However, studies [140–142] 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 [140, 141], 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 [140], by trying out different K values over a small amount of testing dataset, 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 i th dimension is set to 1 if the word belongs to the i th 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 4.2.4). This helps the CRF model tolerate semantically similar API-mention and sentence-context variations, and thus alleviate the out-of-vocabulary issue.

4.2.3.5 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 4.2.3.3). 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 [32], 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 4.2.3.3 (lines 1-2). The details of the CRF classifier are explained in Section 4.2.4. 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

Algorithm 1: Self-training the CRF-based Classifier

Data: A stream of unlabeled sentences *unlabelsents*;
A set of labeled training sentences *te*;
Result: The CRF classifier *l*

```

1 te  $\leftarrow$  human labeled sentences;
2 l = train(te);
3 for S  $\in$  unlabelsents && iterations < M do
4    $\vec{S}$  = feature_extractor(S);
5   (Slabeled, conf) = crf_label(l,  $\vec{S}$ );
6   if conf >  $\alpha$  then
7     te  $\leftarrow$  te  $\cup$  {Slabeled};
8     n = n + 1;
9   end
10  if n > N then
11    l = train(te);
12    iterations ++;
13    n = 0;
14  end
15 end

```

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.

4.2.4 CRF-based Classifier

We continue to use linear-chain Conditional Random Fields (CRF) [27] to determine whether the token is an API mention or a normal word, which is same as S-NER presented in the first section of this chapter. 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 k th token to the current token, e.g., w_{i+1} is the next token to the current token. w_{i-k} denotes the previous k th token to the current token.

Orthographic features. This set of features is similar but slightly different to S-NER, which includes: 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 [140], [141], 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 $\{2, 4, 6, 8, \dots, 14\}$. Note that the Brown Clustering features used here consider the center word and its surrounding words, while the Brown Clustering features used in S-NER only cover the center word.

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

TABLE 4.10: 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%

“e.g”, a simple partial string matching will match the token to the API name like “pandas.core.groupby.GroupBy.transform”, which is not desired.

4.2.5 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.

4.2.5.1 Tool Implementation

We implement web crawlers using Scrapy ¹ to crawl official API names. For the implementation of the linear CRF, we use CRFSuite ², a popular CRF toolkit for sequential labeling. For Brown Clustering, same to S-NER which was covered in Section 4.1, we use Liang’s implementation ³. We learn continuous word embeddings using word2vec ⁴, which contains an efficient open-source implementation of the skip-gram model [30]. We use the K-means implementation from Sofia-ML ⁵ to perform K-means clustering of the continuous word embeddings.

4.2.5.2 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

¹Scrapy, <http://scrapy.org/>

²CRFSuite, <http://www.chokkan.org/software/crfsuite/>

³Brown Clustering, <https://github.com/percyliang/brown-cluster>

⁴word2vec, <https://code.google.com/archive/p/word2vec/>

⁵Sofia-ML, <https://code.google.com/archive/p/sofia-ml/>

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 4.10 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 4.2.3.2. APIs in the inventory are then split into a non-polysemous set and a polysemous set (see Section 4.2.3.3). *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.

4.2.5.3 Model Training

Human labeled sentences for model training are from `Pandas` posts only. The sentence selection and labeling process is described in Section 4.2.3.3. 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, following [140], we tune K values from 500 to 3000, and finally select the combination of 500, 1000, 1500, 2000, 2500, which achieves the best results. These 5 word-embedding clusters and the Brown clusters are used as word-representation features to the CRF (Section 4.2.4).

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 [73], [72], [21]. 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.

4.2.5.4 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.

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 4.11 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, our 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.

TABLE 4.11: Statistics of API Mentions in Testing Dataset

Library	#API Mentions			Total
	standard/deriv ¹	synonym/deriv	polysemy	
Pandas	167	59	182	408
Matplotlib	184	62	189	435
Numpy	88	114	160	362
Total	439	235	531	1,205

¹deriv = derivational form

4.2.5.5 Evaluation metrics

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

4.2.5.6 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 (based on Miler [9]) - Lightweight regular expressions.** We implement the lightweight regular expressions used in Miler [9]. 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 [9] for details).

- **Baseline2 (based on ACE [76]) - Island parsing.** We implement island parsing following the descriptions in Section 4 of [76], except that their parser is based on Java language specification, while ours is based on Python language convention. Note that in [76] authors examine the validity of the extracted APIs using API linking techniques from RecoDoc [10], while such API linking techniques are not used in this baseline. 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 (based on S-NER) - Machine-learning based NER.** We use the software-specific entity recognition tool (S-NER) proposed in Section 4.1 of this Chapter to recognize the API mentions in our testing data. For fair comparison, we use the same set of features used in S-NER, and re-train the model of S-NER with the same set of human labeled sentences for training the CRF model.

4.2.6 Experiment Results and Analysis

We now report experiment results and analyze our findings.

4.2.6.1 Overall Results for All API Mentions

Table 4.12 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 4.12: Comparison of Overall Performance

Method	Precision	Recall	F1-score
Baseline1 (Miler [9])	0.125	0.723	0.213
Baseline2 (ACE [76])	0.633	0.624	0.628
Baseline3 (S-NER)	0.825	0.678	0.744
Our method	0.879	0.872	0.876

TABLE 4.13: API Extraction Performance for Each of the 3 Studied Libraries

Method	Pandas			Matplotlib			Numpy		
	Prec.	Rec.	F1	Prec.	Rec.	F1	Prec.	Rec.	F1
Baseline1 (Miler [9])	0.153	0.791	0.257	0.107	0.689	0.180	0.111	0.675	0.191
Baseline2 (ACE [76])	0.640	0.615	0.627	0.611	0.622	0.617	0.617	0.645	0.631
Baseline3 (S-NER)	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

Comparison with Baseline1: we observe almost the same performance result as that of Miler [9] 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. 4.5, which is common in Stack Overflow discussions.

Comparison with Baseline3: The Baseline3, i.e., machine-learning based software-specific named entity recognition, 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 the Baseline3 (S-NER) can be attributed to the following reasons: 1) The Baseline3 aims to recognize a broad category of software-specific entities, making it difficult to build a gazetteer with good coverage of APIs of different programming languages. By comparison, the work in this section is more focused on disambiguating the sense of a common word and the sense of an API. The polysemous APIs of Python libraries are used as a case study. The gazetteer features designed in this section are observed to be very useful (as will be shown in Section 4.2.6.3), while the gazetteer features have almost no use in the Baseline3. 2) The context feature of our API recognition method is stronger than that of S-NER. We use of unsupervised word representations as compound semantic context features in this section. 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 (i.e., the center word), and less on context features. As a result, the Baseline3's improvement on recall over the island parsing is moderate. 3) 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 the Baseline3.

4.2.6.2 Results for Each of the Studied Libraries

Table 4.13 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 0.749 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 transferred 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*.

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.

4.2.6.3 Feature Ablation

We ablate one kind of feature(s) at a time from our full feature set (see Section 4.2.4) and study the impact of different kinds of features on the API extraction performance. Table 4.14 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

TABLE 4.14: 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

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.

4.2.6.4 The Impact of Self-Training

When training the CRF model, we perform 10 iterations of self-training. Figure 4.7 shows the change of the F1-score for extracting all API mentions after each iteration. We can 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

FIGURE 4.7: The Impact of Self-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.

4.2.7 Discussion

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

4.2.7.1 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 [32]. 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.

4.2.7.2 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, we have 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 all annotators 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.

4.2.8 Summary

In this section, we address 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 (which will be explored in [Chapter 5](#)), API search, and API-related issue-solution mining in software engineering social content.

Chapter 5

Connecting Linkable Software Engineering Knowledge

In this chapter, we develop linking techniques to automatically connect linkable software engineering knowledge. At **document-level**, we aim to link semantically relevant knowledge units (Section 5.1). And at **entity-level**, we link an API mention in a sentence to its formal form appeared in the API documentation (Section 5.2).

5.1 Predicting Semantically Linkable Knowledge

— Connection of Knowledge at Document-level *

5.1.1 Introduction

Following the definition of *knowledge unit* in Chapter 3, we consider a question, its answers and all the comments on Stack Overflow as a knowledge unit regarding some programming-specific issues. The knowledge contained in one unit is likely to be related to knowledge in other units. As studied in Chapter 3, when asking a question or providing an answer in Stack Overflow, users reference existing questions and answers that contain relevant knowledge by URL sharing, which is strongly encouraged

*This section is based on our ASE-2016 conference paper: “Predicting Semantically Linkable Knowledge in Developer Online Forums via Convolutional Neural Network”.

by Stack Overflow ¹. Through URL sharing, a network of *linkable knowledge units* has been formed over time.

Recall that unlike linked pages on Wikipedia that follows the underlying knowledge structure, questions and answers are specific to individual's programming issues, and URL sharing in Q&As is opportunistic, because it is based on the community awareness of the presence of relevant questions and answers. As shown in Chapter 3, the structure of the knowledge network that URL sharing activities create is scale free. A scale free network follows a power law degree distribution, which can be explained using preferential attachment theory [67], i.e., “the rich get richer”. On one hand, this means that a small proportion of the knowledge units is attracting a large proportion of users' attention. On the other hand, this means that a large amount of knowledge units in the long tail of the power law distribution are mostly under linked.

To mitigate this issue, Stack Overflow recommends related questions when users are viewing a question. Stack Overflow's recommendation of related questions is essentially based on lexical similarity of word overlap between questions ². However, linkable knowledge units often share few words in common (i.e., lexical gap) due to two main reasons. First, users could formulate the same question in many different ways. For example, Table 5.1 shows two duplicate questions from Stack Overflow. The question (Id 510357) has been recognized as a duplicate to another question (Id 19477465) by the Stack Overflow users, because they can be answered by the same answer. However, literally, we can hardly see common words and phrases between these two questions. Second, two questions could discuss different but relevant knowledge. For example, Table 5.2 shows two directly linkable questions, one asks “iterating over a stack (reverse list)” while the other asks “best way to check if a list is empty”. Again, the two questions share few common words, but they embody strongly relevant knowledge (stack iteration and list empty checking). Furthermore, questions may discuss relevant knowledge but not directly for solving the questions, as the examples in Table 5.3 shows. As such questions discuss indirectly linkable knowledge, they are unlikely to share many common words. Traditional word representations (e.g., BM25 [115],

¹How do I ask a good question? <http://stackoverflow.com/help/how-to-ask>

²How are related questions selected? <http://meta.stackexchange.com/q/20473>

TABLE 5.1: An Example of Duplicate Knowledge Units from Stack Overflow

<p>Question Id 510357 (marked as <i>duplicate</i> to 19477465)</p> <p>Title: Python read a single character from the user</p> <p>Body: Is there a way of reading one single character from the user input? For instance, they press one key at the terminal and it is returned (sort of like <code>getch()</code>). I know there's a function in Windows for it, but I'd like something that is cross-platform.</p>
<p>Question Id 19477465</p> <p>Title: Get python program to end by pressing anykey and not enter</p> <p>Body: How can I get my Python program to end by pressing any key without pressing enter. So if the user types "c", the program should automatically end without pressing enter. My code so far: <code>print("Hi everyone! This is just a quick sample code I made")</code> <code>print("Press anykey to end the program.")</code></p>

TABLE 5.2: An Example of Directly Linked Knowledge Units from Stack Overflow

<p>Question Id 4547310</p> <p>Title: Iterating over a stack (reverse list), is there an <code>isempty()</code> method?</p> <p>Body: What's the best way to iterate over a stack in Python? I couldn't find an <code>isempty</code> method, and checking the length each time seems wrong somehow.</p>
<p>Question Id 53513</p> <p>Title: Best way to check if a list is empty</p> <p>Body: For example, if passed the following: <code>a = []</code> How do I check to see if a is empty?</p>

TABLE 5.3: An Example of Indirectly Linked Knowledge Units from Stack Overflow (both have an link to 53513)

<p>Question Id 32831543</p> <p>Title: How to check if a nested list has a value</p> <p>Body: I have a nested list and I want to check if an item has a value or not. Not really sure how to describe it, so basically, how do I get this to work?</p>
<p>Question Id 1115313</p> <p>Title: Cost of <code>len()</code> function</p> <p>Body: What is the cost of <code>len()</code> function for Python built-ins? (list/tuple/string/dictionary)</p>

LDA [113]) cannot reliably recognize many cases of potentially linkable knowledge units when lexical gaps exist between them.

As shown in the above examples, knowledge units can be linkable for different purposes. Our empirical study in Chapter 3 on the knowledge network in Stack Overflow confirms this phenomenon. Being able to classify different classes of linkable knowledge units would support more targeted information needs when users search or explore the linkable knowledge. For example, duplicate questions allow users to understand a

problem from different aspects, directly-linkable questions help explain concepts or sub-steps of a complex problem, while indirectly-linkable questions provide extended knowledge. However, such multiclass semantic relatedness in software engineering knowledge is not considered in existing work [16], [117], which focuses on only binary classification, i.e., related or not. Furthermore, existing work heavily relies on human-engineered word- and document-level features to train the classifier, which are hard to design for informal user-generated content in Stack Overflow.

In this section, we propose a novel approach for predicting *multiclass semantically linkable knowledge units* in Stack Overflow. Inspired by the observations in Chapter 3, we consider four classes of semantic relatedness: duplicate, direct linkable, indirectly linkable, and isolated. To overcome the lexical gap issue, our approach recognizes and quantifies semantic relatedness between knowledge units using word embeddings [30], [33] and Convolutional Neural Network (CNN) [34], which are the state-of-the-art deep learning techniques for capturing word-level and document-level semantics for Natural Language Processing (NLP) tasks. Furthermore, our approach does not rely on human-engineered features to classify semantic relatedness. Instead, we collect large amounts of different types of user-created knowledge-unit links (duplicate, direct-linked, indirect-linked) from Stack Overflow to train the CNN to automatically learn the most informative word- and document-level features to classify semantic relatedness between knowledge units.

We conduct extensive experiments to compare our deep-learning based approach with the baseline methods using traditional word representations (TF-IDF) and the human-engineered features for determining semantic similarity. Our results show that: 1) our approach significantly outperforms the baseline methods, and performs much more consistently for different classes of semantic relatedness; 2) both word embeddings and CNN help improve the performance of multiclass classification of linkable knowledge units. CNN plays a more important role for predicting duplicate, directly linkable, and isolated knowledge unit, due to its capability of capturing document-level semantic features, while word embeddings are good at predicting indirectly linkable knowledge units; 3) domain-specific training corpus help improve the performance of deep-learning techniques for specific tasks.

5.1.2 The Approach

This section formulates our research problem and describes the technical details of our approach.

5.1.2.1 Problem Formulation

If the two knowledge units are semantically related, we consider them as *linkable*. We further predict the types of the relatedness between the two knowledge units. Based on our analysis on the purposes of URL sharing in Chapter 3, we define four types of the relatedness between the two knowledge units:

- *Duplicate*: The two knowledge units discuss the same question in different ways, and can be answered by the same answer.
- *Direct link*: One knowledge unit can help solve the problem in the other knowledge unit, for example, by explaining certain concepts, providing examples, or covering a sub-step for solving a complex problem.
- *Indirect link*: One knowledge unit provides related information, but it does not directly answer the question in the other knowledge unit.
- *Isolated*: The two knowledge units are not semantically related.

We formulate our task as a *multiclass classification* problem. Specially, given two knowledge units, our task is to predict whether the two knowledge units have one of the above four types of relatedness.

5.1.2.2 Overview of Main Steps

In NLP tasks, words are usually represented as vectors. We use *word embeddings* (distributed representations of words in a vector space) [30, 31] as word representations, because word embeddings require only large amounts of unlabeled text to train, and have been shown to be able to capture rich semantic and syntactic features of words.

FIGURE 5.1: Overview of the Main Steps

We use continuous skip-gram model [33] (the Python implementation in Gensim [143]) to learn domain-specific word embeddings from large amounts of software engineering text from Stack Overflow questions and answers. The output is a dictionary of word embeddings for each unique word.

Word embeddings encode word-level semantics. To predict semantic relatedness between knowledge units, we need to capture semantics at sentence and knowledge unit level. To this end, we resort to convolutional neural network (CNN) [34]. Given two knowledge units, they are first converted into word vector representations by looking up the word-embeddings dictionary, which are then fed into a CNN to produce a feature vector for each knowledge unit. The similarity of the two knowledge units are measured by the cosine similarity of their feature vectors. Based on the similarity score, the relatedness of the two knowledge units are classified as one of the four classes defined in Section 5.1.2.1.

To train the CNN, we collect a set of *user-linked knowledge-unit pairs* from Stack Overflow for each type of relatedness: duplicate, direct link, indirect link, and isolated. This set of training data is fed into the CNN and the training is guided by the *cosine loss* of the relatedness of the knowledge-unit pairs against the user-linked relatedness. During the training process, the CNN automatically learns the most informative word and sentence features for predicting multiclass linkable knowledge units.

FIGURE 5.2: Example of Word Embedding

5.1.2.3 Learning Word Representations

As also mentioned in Previous Chapters, distributed word representations assume that words appear in similar context tend to have similar meanings [144]. Therefore, individual words are no longer treated as unique symbols, but mapped to a dense real-valued low-dimensional vector space. Each dimension represents a latent semantic or syntactic feature of the word. Semantically similar words are close in the embedding space.

Word embeddings are unsupervised word representations, which requires only large amounts of unlabeled text to learn. In this work, we collect body content of questions and answers from Stack Overflow as software engineering text. The data preprocessing is the same described in Chapter 4. To train word embeddings, we continue to use continuous skip-gram model with negative sampling proposed by Mikolov et al. [30], [33]. Continuous skip-gram model learns the word embedding of a center word (i.e., w_i) that is good at predicting the surrounding words in a context window of $2k + 1$ words.

Figure 5.2 shows some examples of domain-specific word embeddings we learn from Stack Overflow *java* text, visualized in a two-dimensional space using the t-SNE dimensionality reduction technique [145]. Semantically close words, such as *JPanel*,

FIGURE 5.3: Architecture of the Proposed CNN

JButton, *JFrame* and *JLabel* which belong to GUI component are close in the vector space.

The output of the model is a dictionary of words, each of which is associated with a vector representation. Given a knowledge unit, its body content (after proper pre-processing) will be converted into a knowledge-unit vector by looking up the dictionary of word embeddings and concatenating the word embeddings of words comprising the body content. Let $wv_i \in \mathbb{R}^d$ be the d -dimensional word vector corresponding to the i -th word in a knowledge unit. The knowledge unit of n words is represented as, $kuv = wv_1 \oplus wv_2 \oplus \dots \oplus wv_n$, which is used as input to the CNN.

5.1.2.4 The CNN Architecture

Figure 5.3 presents the architecture of our CNN. Our CNN takes as input a knowledge-unit vector $kuv = wv_1 \oplus wv_2 \oplus \dots \oplus wv_n$, and captures the most informative word and sentence features in the knowledge unit for determining semantic relatedness. Treating the knowledge-unit vector as an “image”, we perform convolution on it via linear filters. Because each word is represented as a d -dimensional vector, we use filters with widths k equal to the dimensionality of the word vectors (i.e., $k = d$). Let $wv_{i:i+h-1}$ refer to the concatenation vector of h adjacent words $w_i, w_{i+1}, \dots, w_{i+h-1}$. A convolution operation involves a *filter* $w \in \mathbb{R}^{hk}$ (a vector of $h \times k$ dimensions) and a *bias term* $b \in \mathbb{R}^h$, which

is applied to h words to produce a new value $o_i \in \mathbb{R}$:

$$o_i = w^T \cdot wv_{i:i+h-1} + b \quad (5.1)$$

where $i = 1 \dots n - h + 1$, and $\cdot$ is the dot product between the filter vector and the word vector. This filter is applied repeatedly to each possible window of h words in the sentences (zero padding where necessary) in the knowledge unit (i.e., $wv_{1:h}, wv_{2:h+1}, \dots, wv_{n-h+1:n}$) to produce an *output sequence* $o \in \mathbb{R}^{n-h+1}$, i.e., $o = [o_1, o_2, \dots, o_{n-h+1}]$. We apply the non-linear *activation function* *ReLU* [146] to each o_i to produce a *feature map* $c \in \mathbb{R}^{n-h+1}$ where $c_i = \text{ReLU}(o_i) = \max(0, o_i)$.

To capture features of phrases of different length, we vary the *window size* h of the filter, i.e., the number of adjacent words considered jointly. In our CNN, we use filters of 5 different window size (1, 3, 5, 7, 9), which can capture features of different n-grams respectively. For each window size, we use 128 filters to learn complementary features from the same word windows. The dimensionality of the feature map generated by each filter will vary as a function of the knowledge-unit length and the filter's window size. Thus, a pooling function should be applied to each feature map to induce a fixed-length vector. In this work, we use *l-max pooling*, which extracts a scalar (i.e., a feature vector of length 1) with the maximum value in the feature map for each filter. Together, the outputs from each filter can be concatenated into a top-level feature vector, denoted as fv_{ku} . This feature vector would capture the most informative 1, 3, 5, 7 and 9-grams in the input knowledge unit.

Given two knowledge units ku_x and ku_y , the CNN outputs their respective feature vectors fv_x and fv_y . For this pair of feature vectors fv_x and fv_y , the CNN computes a similarity score between fv_x and fv_y in the last layer, using the cosine similarity, i.e.,

$$\text{relatedness}(ku_x, ku_y) = \frac{fv_x \cdot fv_y}{\|fv_x\| \|fv_y\|}$$

In the training phase, the computed similar score will be used to compute the mean square error against the ground-truth similar score for the given pair of knowledge units (see Section 5.1.2.5). In the prediction phase, the computed similar score will be used

to determine the class of semantic relatedness of the given two knowledge units. Because the computed similarity score is a continuous value, we need to bin the similarity score as four discrete classes of semantic relatedness between the two knowledge units: $[0, 0.25)$ as isolated, $[0.25, 0.5)$ as indirect link, $[0.5, 0.75)$ as direct link, and $[0.75, 1]$ as duplicate.

5.1.2.5 Training the CNN

Training of the proposed CNN follows supervised learning paradigm. According to our task objective, the training data must consist of sufficient examples of four types of related knowledge-unit pairs, i.e., duplicate, direct link, indirect link and isolated, so that the CNN can learn to capture informative word and sentence features for classifying knowledge-unit relatedness.

Fortunately, URL sharing activities by Stack Overflow users create all types of the needed relatedness. User-created links between knowledge units are stored in the *post_links* table. We parse the *post_links* to generate the training dataset as follows. First, we randomly select pairs of user-linked knowledge-unit pairs. If the *LinkTypeId* is 3 (i.e., duplicate posts), we mark the selected pair as an instance of duplicate knowledge units. Otherwise, the selected pair is an instance of direct-link. Second, we randomly select pairs of knowledge units that are only transitively linked (i.e., the shortest path between the knowledge units is at least 2). These pairs of knowledge units are instances of indirect-link knowledge units. Finally, we randomly select pairs of knowledge units that do not have links in the *post_links* table. These pairs are instances of isolated knowledge units. We select equal number of instances for different types of relatedness.

Let the tuple $\langle ku_x, ku_y, simValue \rangle$ be a pair of knowledge units in the training dataset *TD*, and *simValue* is the ground-truth similarity score between ku_x and ku_y . In this work, we set 0.15, 0.40, 0.65 and 0.8 as the ground-truth similar score for the four classes: isolated, indirect link, direct link and duplicate, respectively. The CNN with the current parameter set θ computes a similarity score between the knowledge units, i.e., $relatedness(ku_x, ku_y)$. The training objective is to minimize the mean-squared error of the computed similarity score and the ground-truth similarity score (i.e., *cosine*

$loss$) with respect to θ :

$$\theta \mapsto \frac{1}{|TD|} \sum_{TD} (simValue - relatedness(ku_x, ku_y))^2$$

We add the $l2$ norm regularization loss to the mean-squared error data loss. The $l2$ norm constraint penalizes large weight parameters. It helps avoid model overfitting, because no input dimension can have a very large influence on the predicted probabilities all by itself. The parameters to be estimated include: the convolution filters w and the bias terms b (see Eq. 5.1). We solve the optimization problem using Adam update algorithm [147]. Once we learn the CNN parameters, the CNN can be used to predict the semantic relatedness of an unseen pair of knowledge units.

5.1.3 Experiment

We conduct a set of experiments to evaluate the effectiveness of our approach, compare it again a well-designed baseline, and investigate the impact of domain-specific word embeddings. As our approach relies on deep learning techniques, we also report the training cost of our approach.

5.1.3.1 Baseline Building

As our task is essentially to predict relatedness between two pieces of text, we design two baselines, each of which is a multiclass SVM classifier with different textual features, i.e., traditional Term Frequency (TF) and Inverse Document Frequency (IDF) versus word embeddings.

Feature Extraction with TF and IDF. TF and IDF are widely used in predicting textual similarity with cosine distance [15]. In this baseline, we define in total 36 features based on the TF and IDF values of the words in a pair of knowledge units. That is, the pair of the knowledge units is represented as a 36-dimensional feature vector to be used as input to the SVM classifier.

Given a knowledge unit KU_x , we split its text into three sub-documents, denoted as C_x^i ($1 \leq i \leq 3$), which correspond to question title ($i = 1$), question body ($i = 2$), and question title plus question body plus body of all answers ($i = 3$). Let $\langle KU_x, KU_y \rangle$ be a pair of knowledge units.

For TF-based features, we first obtain a vocabulary $V_{x,y}^{i,j}$ with all words in a combination of different sub-documents of the two knowledge units, i.e., $V_{x,y}^{i,j} = \{w \mid w \in C_x^i \cup C_y^j \ (1 \leq i, j \leq 3)\}$. The TF value of each word in the vocabulary $V_{x,y}^{i,j}$ is then computed. These word TF values are used to convert the sub-document C_x^i of the knowledge unit KU_x into a vector representation v_x^i . The dimensionality of the v_x^i is the size of the vocabulary, and each dimension is the TF value of the word in the sub-document C_x^i , or 0 otherwise. We compute the cosine similarity of the two vectors v_x^i and v_y^j to measure the similarity of the sub-document C_x^i and C_y^j of the knowledge unit KU_x and KU_y , respectively. As each knowledge unit has three sub-documents, i.e., $1 \leq i, j \leq 3$, we obtain 3×3 (9) TF-based features for a pair of knowledge units.

For IDF-based features, we first combine the respective sub-documents C_x^i of all knowledge units in a dataset as a corpus C^i . The IDF value of a word w in the corpus C^i is then computed, denoted as $idf_i(w)$. We measure the IDF-based similarity between the sub-documents C_x^i of the knowledge unit KU_x and the sub-document C_y^j of the knowledge unit KU_y as:

$$SimIDF_k(C_x^i, C_y^j) = \sum_{w \in C_x^i \cap C_y^j} idf_k(w)$$

i.e., the sum of the IDF value of the common words of C_x^i and C_y^j in the corpus C^k . As $1 \leq i, j, k \leq 3$, we obtain $3 \times 3 \times 3$ (27) IDF-based features for a pair of knowledge units.

Feature Extraction with Word Embeddings. The semantic information of a piece of text can be captured by taking the mean of the word embeddings of the words comprising the text [148]. Following this treatment, we compute the mean of the word

embeddings of the words in a knowledge unit KU_x (including question title plus question body plus body of all answers) as the word embedding of the knowledge unit, i.e.,

$$wv(KU_x) = \frac{1}{n} \sum_{w \in B_x} wv(w)$$

where B_x is the bag of words of the KU_x and n is the size of B_x . A pair of knowledge units KU_x and KU_y is then represented as the mean of the two knowledge-unit word embeddings, i.e.,

$$wv(KU_x, KU_y) = \frac{1}{2}(wv(KU_x) + wv(KU_y))$$

which will be used as input to the SVM classifier. In this work, we use 200-dimensional word vector.

Multiclass SVM Classifier. We develop a multiclass SVM classifier to predict the relatedness of the two knowledge units. Based on the feature vector used (TF-IDF based or word embedding based as described above), we have two baselines: Baseline1 (TF-IDF+SVM) and Baseline2 (WordEmbed+SVM). The two baseline SVM classifiers are trained using the same dataset as that for training our approach. As our task is a multiclass classification problem, we use RBF kernel [149] in the SVM model. We set the γ parameter of the SVM to $1/k$, k denotes the dimensionality of the feature vector, i.e., $1/36$ for the TF-IDF+SVM baseline and $1/200$ for the WordEmbed+SVM baseline. We use grid search [150] to optimize the SVM parameters.

5.1.3.2 Dataset

Our experimental data used in this section is from Stack Overflow data dump of March 2016: <https://archive.org/download/stackexchange>.

Word Embedding Corpora. From the *posts* table, we randomly select 100,000 knowledge units tagged with “java” as the word embedding corpora. Note that our corpora contains not only questions but also question answers.

Training and test linkable knowledge units. From the *postlinks* table, we randomly select in total 6,400 pairs of knowledge units that are tagged with “java”, i.e., 1,600 pairs for each type of relatedness defined in Section 5.1.2.1 (*Duplicate*, *Direct Link*, *Indirect Link* and *Isolated*). These 6,400 pairs of knowledge units are used as training data. For test data, we select in total 1,600 pairs of knowledge units that are tagged with “java”, i.e., 400 pairs for each type of relatedness. User-created links are considered as ground truth label for the semantic relatedness of the selected pairs of knowledge units. The knowledge units in the test data does not overlap with those in the training data.

5.1.3.3 Evaluation Metrics

We represent the multiclass classification results as a $K \times K$ matrix A , where K is the number of classes (in our work $K = 4$). The rows represent the ground truth labels and the columns represent the predicted labels. The value of the A_{ij} is the number of times that a pair of knowledge units with the ground truth label L_i that is classified as the label L_j . Therefore, the value of A_{ii} is the number of correct classification for the label L_i , $\sum_{1 \leq j \leq K} A_{ij}$ is the number of the ground truth label L_i in a dataset, $\sum_{1 \leq i \leq K} A_{ij}$ is the number of predicted label L_j in a dataset, and $\sum_i \sum_j A_{ij}$ is the number of knowledge-unit pairs in a dataset.

We use the following metrics to compare our approach with the two baselines:

Accuracy is defined as the proportion of numbers of correct classification in a dataset, i.e.,

$$Accuracy = \frac{\sum_i A_{ii}}{\sum_i \sum_j A_{ij}}$$

Precision for a class j is the proportion of knowledge-unit pairs correctly classified as the class j among all pairs classified as the class j . Precision for all classes is the mean of the precision for each class.

$$Precision_j = \frac{A_{jj}}{\sum_{1 \leq i \leq K} A_{ij}}$$

Recall for a class i is the percentage of knowledge-unit pairs correctly classified as the class i compared with the number of ground truth label L_i in the dataset. Recall for all classes is the mean of the recall for each class.

$$Recall_i = \frac{A_{ii}}{\sum_{1 \leq j \leq K} A_{ij}}$$

F1-score for a class i is a harmonic mean of precision and recall for that class. F1-score for all classes is the mean of the F1-score for each class.

$$F1_i = \frac{2 \times Precision_i \times Recall_i}{Precision_i + Recall_i}$$

F1-score evaluates if an increase in precision (or recall) outweighs a loss in recall (or precision). We choose F1-score as the main evaluation metric.

5.1.3.4 Research Questions and Findings

We aim to answer the following research questions:

RQ1: *How much improvement can our approach achieve over the baseline approaches? – Overall Comparison*

Motivation. Our approach uses deep learning techniques (word embeddings and CNN-derived features) to quantify semantic relatedness between two pieces of software engineering text, which is very different from the baseline approaches using traditional TF-IDF word representations and human-engineered features. Answer to this research question will shed light on whether and to what extent deep learning techniques can improve the results of the multiclass classification task on software engineering text.

Approach. We apply our approach and the baseline approaches to the test data, i.e., 1,600 pairs of linkable knowledge units. We compare the accuracy, precision, recall, and F1-score metrics of different approaches.

Result. Table 5.4 and Table 5.5 present the results. We can see that the accuracy of our approach outperforms the *Baseline1* and *Baseline2* by 34.6% and 25.7%, respectively.

TABLE 5.4: Accuracy Comparison

		Accuracy
Baseline1	TF-IDF + SVM	0.625
Baseline2	Word Embedding + SVM	0.669
Our Approach	Word Embedding + CNN	0.841

TABLE 5.5: Precision, Recall and F1-Score of Our Approach and the Baseline Approaches

			Duplicate	Direct Link	Indirect Link	Isolated	Overall
Precision	Baseline1	TF-IDF + SVM	0.585	0.611	0.542	0.767	0.626
	Baseline2	Word Embedding + SVM	0.611	0.560	0.787	0.676	0.659
	Our Approach	Word Embedding + CNN	0.898	0.758	0.840	0.890	0.847
Recall	Baseline1	TF-IDF + SVM	0.81	0.413	0.505	0.773	0.625
	Baseline2	Word Embedding + SVM	0.725	0.433	0.980	0.538	0.669
	Our Approach	Word Embedding + CNN	0.898	0.903	0.773	0.793	0.842
F1-Score	Baseline1	TF-IDF + SVM	0.679	0.493	0.523	0.770	0.616
	Baseline2	Word Embedding + SVM	0.663	0.488	0.873	0.600	0.656
	Our Approach	Word Embedding + CNN	0.898	0.824	0.805	0.849	0.841

Overall (see the last column of Table 5.5), our approach outperforms the *Baseline1* and *Baseline2* in terms of F1-score by 36.5% and 28.2%, respectively. Similar improvement on overall precision and recall can be observed.

Overall, our approach achieves the best performance in all the evaluated metrics by a substantial margin, compared with the two baseline approaches.

RQ2: How effective is our approach in predicting different classes of semantic relatedness, compared with the baseline approaches? – Comparison by Different Classes

Motivation. Different from existing work on binary text classification, our approach is for multiclass text classification. Different classes of semantic relatedness may exhibit different word and sentence features, which may affect the effectiveness of different semantic similarity measures. We would like to investigate the advantages and disadvantages of our approach and the baseline approaches for predicting different classes of semantic relatedness.

Approach. We compare the precision, recall and F1-score for each class of relatedness (i.e., duplicate, direct link, indirect link, and isolated) by our approach and the two baseline approaches (see Table 5.5).

Result. The performance of our approach and the baseline approaches do vary for different classes of relatedness. Our approach performs the best for three of the four classes (i.e., duplicate, direct link and isolated) on all the evaluation metrics, while the Baseline2 performs the best for the indirect link class only on recall and F1-score. In fact, the Baseline2 achieves the extremely high recall (0.980) but low precision (lower than our approach) for the indirect link class. The performance variance of our approach for different classes is small (0.05-0.09 in terms of F1-score), compared with the performance variance of the two baseline methods for different classes (in terms of F1-score, 0.1-0.27 for the Baseline1 and 0.27-0.39 for the Baseline2).

Our approach performs better on more classes of relatedness than the baseline methods (3:1). Our approach also perform more consistently for different classes of relatedness. The Baseline2 is extremely good at recalling indirectly linkable knowledge units with a sacrifice of precision, but its performance vary greatly for different classes of relatedness.

RQ3: What is the impact of word embeddings and CNN on the performance improvement respectively? – Effects of Word Embeddings and CNN

Motivation. Word embeddings and CNN are two deep learning techniques our approach relies on. They help capture semantics at word-level and sentence/document-level respectively. Answer to this research question helps us understand the importance of different levels of semantics for the multiclass text classification task.

Approach. To understand the importance of word embedding, we compare the performance of the Baseline1 (i.e., TF-IDF+SVM) and the Baseline2 (i.e., WordEmbedding+SVM). To understand the importance of CNN, we compare the performance of the Baseline2 (i.e., WordEmbedding+SVM) and our approach (i.e., WordEmbedding+CNN).

Result. Overall, the Baseline2 outperforms the Baseline1 by a small margin (see Accuracy in Table 5.4 and the Overall Column in Table 5.5). This suggests that word embeddings can moderately improve the text classification performance compared with the

traditional TF-IDF word representation. Overall, our approach outperforms the Baseline2 by a substantial margin. This suggests that CNN plays a more important role than word embeddings for improving the multiclass classification performance.

The performance of the Baseline2 is not always better than that of the Baseline1 for different classes of relatedness. For some classes, the Baseline2 has better precision but worse recall than the Baseline1, and vice versa for other classes. In terms of F1-score, the two baselines are almost the same for duplicate and direct link classes, while the Baseline2 is significantly better than the Baseline1 for indirect link class, but significantly worse for isolated classes. This suggests that word-level semantics encoded in word embeddings may not be appropriate for determining all classes of relatedness. For isolated knowledge units, traditional TF-IDF representation which is sensitive to lexical gap performs better than word embeddings. Taking the mean of word embeddings as knowledge-unit representations may blur the semantic distinction between the knowledge units, which helps the Baseline2 recall indirectly linkable knowledge units, but degrades the performance of the Baseline2 for isolated knowledge units.

In contrast, our approach consistently outperforms the Baseline2 on all the evaluation metrics for different classes of relatedness, except the recall and F1-score for indirectly linkable knowledge units. This suggests that word-level semantics are especially useful for determining semantic similarity of indirectly linkable knowledge units. As indirectly linkable knowledge units may not exhibit semantic similarity at sentence/document level, considering sentence/document-level semantics by the CNN could rule out false-negative links, which degrades its recall.

Both word embeddings and CNN help improve the performance of multiclass text classification, but CNN has a bigger impact than word embeddings. Word-level semantics are especially useful for predicting indirectly linkable knowledge units, while sentence/document-level semantics plays more significant role for predicting duplicate, directly linkable and isolated knowledge units.

RQ4: How sensitive is our approach to word embeddings learned from different corpus? – Impact of Domain-Specific Word Embeddings

Motivation. In this work, we predict linkable knowledge in developers' online forum. Therefore, we learn word embeddings from Stack Overflow text which is representative of the ways people ask/answer questions and the vocabulary people use. We would like to investigate whether and to what extent word embeddings learned from a different corpora affect the performance of our approach. This will help us understand the importance of suitable corpus for a software-specific machine learning task.

Approach. We collect a corpus of Wikipedia Web pages from the Wikipedia data dump: <https://dumps.wikimedia.org/enwiki/latest/>. The number of sentences and the size of the vocabulary of the Wikipedia corpus is comparable to that of the corpus of the 100,000 knowledge units from Stack Overflow. We learn word embeddings from the Wikipedia corpus and use it to train the CNN subsequently. We compare the performance of the two CNNs, one trained with Stack Overflow word embeddings, and the other trained with Wikipedia word embeddings.

TABLE 5.6: Performance of Our Approach with Word Embeddings Learned from Different Corpus

	Accuracy	Precision	Recall	F1-score
General Corpus From Wikipedia	0.770	0.790	0.776	0.777
Domain-specific Corpus From Stack Overflow	0.841	0.847	0.842	0.841

Result. Table 5.6 presents the accuracy, and the overall precision, recall and F1-score of our approach with domain-specific word embeddings (i.e., Stack Overflow) versus general word embeddings (i.e., Wikipedia). General word embeddings degrade the performance of our approach, compared with domain-specific word embeddings. However, the degrade is moderate, in terms of accuracy, precision, recall and F1-score by 7%, 5%, 7% and 7%, respectively. Furthermore, although the performance of our approach degrades with general word embeddings, it still outperforms the two baseline methods (at least 11% on all the evaluation metrics).

Our approach demonstrates certain level of reliability even with word embeddings learned from a general corpus that is completely different from Stack Overflow discussions. However, suitable domain-specific word embeddings leads to better performance.

RQ5: What is the time cost for training the underlying deep learning models? – Training Cost

Motivation. The underlying word embeddings and CNN models need to be trained before they can be used for prediction task. Training of word embeddings and the CNN model is done only once offline. After model training, using the model to predict the relatedness of a pair of knowledge units takes negligible time. Understanding the training time cost helps us understand the practicality of our approach.

Approach. We record the start time and the end time of the program execution to obtain the training time cost of the word embeddings and the CNN model. The experimental environment is an Intel(R) Core(TM) i7 2.5 GHz PC with 16GB RAM running Windows7 OS (64-bit).

Result. For learning word embeddings model, the 100,000 knowledge units contain 23,759,119 words (as bags of words), and the vocabulary size (i.e., the number of unique words) is 434,836. It takes about 15 minutes to analyze the text of these 100,000 knowledge units to learn the word embeddings. For training the CNN model, it takes about 14 hours for the CNN model to achieve the loss convergence ($< e^{-3}$).

Training of the word embeddings model and the CNN model can be done efficiently offline and only need to be done once. Our approach is practical for large dataset.

5.1.4 Discussion

This section presents the qualitative analysis of some examples to illustrate the capability of our approach and discusses threads to validity of our experiments.

5.1.4.1 Qualitative Analysis

Table 5.7 presents one example for each class of linkable knowledge unit and the classification results by our approach and the two baselines. The first example is a pair of duplicate questions. However, the two questions do not have many words in common.

The two baseline methods classify them as indirectly linkable knowledge units. In contrast, our approach can capture semantic similarity between terms like “standard input / output” and “System.out.println()”. This helps our approach classify the two questions as duplicate.

In the second example, the two knowledge units contain directly relevant knowledge, i.e., the “GUI repaint() problem” is directly relevant to “text not displaying correctly” problem. Unfortunately, the two knowledge units share few words in common, which makes the baseline methods classify them as isolated. Our approach can capture the semantic relatedness between the two technical problems, and thus correctly classify the two knowledge units as direct link.

In the third example, the two indirectly linked questions share some common words. Based on these overlap words, the Baseline1 which essentially relies on lexical similarity of overlapping words classifies the two questions as duplicate. The Baseline2 which considers word-level semantics by word embeddings makes a better judgment, classifying them as isolated. Our approach makes the most accurate prediction by taking into account not only word-level but also document-level semantics. Similarly, in the fourth example, the Baseline1 makes the least appropriate prediction based on overlapping words, the Baseline2 makes a better judgment based on word-level semantics, and our approach makes the most accurate prediction.

5.1.4.2 Error Analysis

We also analyze the cases in which our approach makes the wrong prediction. We find that many cases that fail our approach involve knowledge units whose essential information is presented as a code snippet (e.g., <http://stackoverflow.com/q/5985912>), or an image (e.g., <http://stackoverflow.com/q/4382178>). In this work, we remove code snippets and images during preprocessing. In the future, we could incorporate code snippets and images into our approach. Incorporating image semantics into our approach would be relatively easy because CNN is originally invented for image classification. Incorporating code semantics into our approach could be a challenging task. A recent work by Mou et al. [151] proposes to use CNN to encode

TABLE 5.7: Examples of Different Classes of Semantically Linkable Knowledge Units

	Knowledge Unit1	Knowledge Unit2	Prediction
Duplicate	Question Id : 2169330 Title : Java, JUnit - Capture the standard input / Output for use in a unit test Body : I'm writing integration tests using JUnit to automate the testing of a console based application. <i>./...</i> I only need to have the test execute and verify the the output is what is expected given the input . <i>./...</i>	Question Id : 10561540 Title : In JUnit testing is it possible to check the method System.out.println() Body: Is it possible to check, through JUnit testing , if the method System.out.println("One, Two") , actually prints One, Two?	Our approach : Duplicate Link Baseline1 : Indirect Link Baseline2 : Indirect Link
	Question Id : 1775858 Title: Text in Label not displaying correctly with setText method Body : I'm trying to set the text in a label dynamically by calling the setText method whenever a button is clicked. <i>./...</i> that are passed to the setText method aren't visible on the screen when the submit button is clicked until I click on the window and drag to resize it. <i>./...</i> On a PC, the strings are visible but obscured until I resize the window . <i>./...</i> On the Mac, the strings appear as intended, however, they're obscured until the window is resized. What am I missing? Answer : Calling validate() on the Frame as mentioned here (Link to Question Id: 369823) solved the problem.	Question Id : 369823 Title : Java GUI repaint() problem? Body: I have a JFrame . This JFrame contains a JButton . I click the JButton and 10 JTextField s are created. the problem: I cannot see them until "I force a repaint() " by resizing the window . Only then do I see the JTextField s created. CODE: <i>./...</i> Answer: Container.add API docs sayeth: Note: If a component has been added to a container that has been displayed, validate must be called on that container to display the new component . If multiple components are being added, you can improve efficiency by calling validate only once. <i>./...</i>	Our approach : Direct Link Baseline1 : Isolated Baseline2 : Isolated
Indirect Link	Question Id :465099 Title: Best way to build a Plugin system with Java Body: How would you implement a Plugin-system for your Java application? Is it possible to have an easy to use (for the developer) system which achieves the following: Users put their plugins into a subdirectory of the app. The Plugin can provide a configuration screen If you use a framework, is the license compatible with commercial development?	Question Id :6973820 Title: plugin base pattern for web app Body: I would like in a web application to be able to add some functionality. Is there any pattern for a web application to be able to add plugin ? Every plugins could be a jar file and the user could select the plugin it want to use. Does somebody have information about this kind of action?	Our approach : Indirect Link Baseline1 : Duplicate Link Baseline2 : Isolated
	Question Id :2438387 Title : Hibernate - How to store java.net.URL into a database through Hibernate Body : I have a field URL countryURL; in a Country class. I want to store its data into a COUNTRY table in a database through Hibernate. Which Hibernate type I should use in the hibernate mapping file <i>./...code.../</i> It is not excepting string and text type.	Question Id : 644814 Title: Does Hibernate EntityManager include Core? Body: I'm using Hibernate's implementation of JPA. <i>./...</i> My question is, does Hibernate EntityManager depend on or use the Hibernate Core code? <i>./...</i> I want to know if EntityManager uses Core and thus has this fix.	Our approach : Isolated Baseline1 : Direct Link Baseline2 : Indirect Link

the program ASTs for program analysis tasks. However, their approach is not directly applicable to code snippets in Q&A discussion, which is usually incomplete and cannot be compiled. We plan to tackle this challenge as our future work.

5.1.4.3 Threats to Validity

There are several threats that may potentially affect the validity of our experiments. Threats to internal validity relate to errors in our experimental data and tool implementation. We have double checked our experimental data and tool implementation. We have also manually checked the selected knowledge units in our dataset to ensure that they are really tagged with "java" and have the right types of knowledge-units links. Threats to external validity relate to the generalizability of our results. In this study, we use a medium-size training and test dataset (100,000 knowledge units for word embedding learning, 6,400 pairs of knowledge units for CNN training, and 1,600 pairs of knowledge units for testing). This allows us to perform some manual analysis to understand the capability and limitations of our approach. In the future, we will reduce this threat by extending our approach to larger word embeddings corpus and more knowledge-unit pairs for training and testing.

5.1.5 Summary

In this section, we propose a novel deep-learning based approach for predicting multiclass semantically linkable knowledge units in Stack Overflow. Our approach can predict four types of semantic relatedness, *duplicate link*, *direct link*, *indirect link* and *isolated*. At word level, our approach adopts the state-of-the-art distributed word representations (i.e., word embedding) to encode word semantics in dense low-dimensional real-valued vectors. At document level, we train a CNN to automatically learn the most informative word and sentence features for classifying the semantic relatedness between two knowledge units. The training of the CNN is guided by sufficient examples of different types of semantically linkable knowledge units. Due to the adoption of word-level and document-level semantics, our approach is robust to the lexical gap (i.e., sharing few words in common) between linkable knowledge units. Our experiments confirm the effectiveness and consistency of our approach for predicting multiclass semantically linkable knowledge units, compared with the well-designed baselines using TF-IDF word representations and human-crafted word and sentence/document features. In the future, we will enhance our approach by incorporating image and code-snippet

semantics into our framework. We will also develop automated tool to help developers search and explore different types of linkable knowledge in Stack Overflow.

5.2 Fine-grained API Linking

– Connection of Knowledge at Entity-level *

In the previous section, we connect linkable knowledge at document-level based on the semantic relatedness between knowledge units. In this section, we focus on connecting software engineering knowledge at entity-level.

5.2.1 Introduction

Textual content can be ambiguous because human language is not exact, which is why we need entity disambiguation (or entity linking) [120] in general domains. For example, the word “Mac” can refer to a computer brand, McDonald’s burger, Migration Authorization Code, etc. This is also the case for API mentions in text. It has been pointed out in Section 4.2 in Chapter 4, Stack Overflow users rarely write the fully qualified name of an API, and instead, the unqualified informal name has been widely used, particularly the simple name. The partial qualified name and simple name of an API can be ambiguous [10, 77], e.g., the simple name of a method can have multiple mappings in different classes of the same library or of different libraries. In [77], Subramanian et al. observe 89% of the Java unqualified method names collide. The ambiguity in API mentions leads to the fact that even after we recognize a word as a true API mention, we may still not know which specific API it is referring to. The process of resolving the ambiguities in API mentions is known as fine-grained API linking [10, 76, 77].

In this section, we propose an API linking approach to link a recognized API mention appeared in Stack Overflow textual discussions to its unique formal form written in the official API documentation, based on our API recognition results (our API recognition approach has been covered in Section 4.2 in Chapter 4). The task of API linking

*This section is based on our ongoing work: “API Recognition and Linking in Developer Online Forums”.

FIGURE 5.4: Illustrating the Task of API Linking

is illustrated in Figure 5.4. In this example, after we recognize the API’s simple name “apply” in the sentence shown in Figure 4.5 in Section 4.2, we aim to further determine which “apply” it is, since there are more than one method named “apply”.

In general domains, entity linking is an important step to make sense of the unstructured textual data. It can facilitate advanced forms of searching in various domains, e.g., linking users’ queries to actual entities is widely used in commercial search engines to improve the user experience. Indeed, entity linking is considered as the building stone for semantic search [3]. By comparison, in the context of software engineering, to move towards our broad goal of developing domain-specific knowledge graph and semantic search, we need to perform linking for software-specific entities. The inherent ambiguous nature of API mentions in natural language text poses the opportunity for conducting domain-specific entity linking in software engineering. API linking can bridge the knowledge in API documentation and the online explanations and examples on Stack Overflow [77]. While we link the knowledge units internal to Stack Overflow in the first section of of this chapter (Section 5.1), we link an API to its formal form in the documentation that is external to Stack Overflow in this section.

The challenge for API linking lies in the various types of ambiguity of API mentions, including declaration ambiguity, external reference ambiguity and synonym ambiguity, which will be further discussed in Section 5.2.2.2. Our API linking approach leverages mention-mention similarity, scope filtering, and mention-entry similarity to

resolve these ambiguities to determine which API entry in the knowledge base the recognized API mention actually refers to. We borrow the terms *mention-mention similarity* and *mention-entry similarity* from the entity linking work done by Liu et al. [2013]. Specifically, mention-mention similarity measures the similarity between recognized API mentions, and mention-entry similarity computes the similarity between an API mention and an entry in the knowledge base. The scope filtering is mainly comprised of a set of Stack Overflow specific information, such as tags, question title and code blocks, to help eliminate the number of linking candidates.

To evaluate our API linking approach, we continue to use the APIs of the three Python libraries used in Section 5.2 of Chapter 4, i.e., *Pandas*, *Matplotlib* and *Numpy*. We populate the knowledge base for linking using all the APIs from the upper 3 libraries, and also from Python standard library (version 2.7 is used in our work), because Python standard APIs are often discussed in many *Pandas*, *Matplotlib* and *Numpy* questions. We examine the linking results of 120 method names that have more than 1 linking candidates in the knowledge base. Experimental results show the effectiveness of our approach.

5.2.2 Background

5.2.2.1 Steps for API Linking

API recognition (or API extraction) is the pre-requisite to API linking, i.e., we must be able to recognize an API mention before we link it. Similar to entity linking (an API is an entity in software engineering) [120], building a fine-grained API linking system involves the following three steps:

1. API recognition. Given a natural language sentence, this step aims to determine if a sequence of characters is an API or not. Note that this step has been done in Section 4.2 in Chapter 4.
2. Knowledge base population. Existing entity linking methods are built on top of a large knowledge base such as Wikipedia [120]. A knowledge base consists

of a set of entities. In the context of API linking, we need a knowledge base that contains information of the studied APIs. Such knowledge base is also been named as *code element index* in the literature [76].

3. API linking, i.e., disambiguating an API mention to its unique form. An API mention can be ambiguous and can be mapped to more than 1 entries in the knowledge base. This step aims to tell precisely which entry of the knowledge base built in Step 2 the recognized API mention from Step 1 is actually referring to.

5.2.2.2 Ambiguities of API Mentions

An API linking system should be able to handle the inherent ambiguities of API mentions. In [2012], Dagenais and Robillard summarize four kinds of ambiguities lying in the supporting documents of the Spring framework written in Java (include both natural language texts and code snippets). In [2014], Subramanian et al. highlight two kinds of ambiguities in the Java code snippets from Stack Overflow. Compared to these two work, we are concerned with linking API mentions in informal natural language texts from Stack Overflow excluding code snippets, and we use Python APIs as a case study, because Python APIs often have no distinctive orthographic features, as mentioned earlier. We discuss the ambiguities of API mentions in informal natural language text that pose challenges to our API linking process as follows.

Declaration Ambiguity. We reuse the term “declaration ambiguity” from [10] and [77]. Fully qualified API names are rarely mentioned in developers’ social discussions on Stack Overflow. Instead, Stack Overflow users often use the simple name or other derivational forms of an API in a post, as shown in Table 4.8 and in Figure 4.5 in Chapter 4. In such cases, it is not an easy task for the machine to automatically and precisely infer the correct linking target.

External Reference Ambiguity. We continue to use the term “external reference ambiguity” same as [10] and [77]. One Stack Overflow knowledge unit can cover knowledge of more than one library. As such, Stack Overflow text may refer to an API declared in an external library but has the same simple name or partially qualified name as one of the APIs in the libraries of focus. For example, *Pandas* is one of our

studied library, Stack Overflow discussions on *Pandas* may refer to an API in the Python Standard Library, or other related libraries.

Synonym Ambiguity. As mentioned earlier in previous chapters, the social nature of online Q&A site introduces synonyms, abbreviations, misspellings (errors), inconsistent annotations, etc. Consequently, the same API is often mentioned in various forms, as shown in Table 4.8, while these writing forms should be linked to the same entry in the knowledge base. The ambiguity of such API mentions is different from the *language ambiguity* defined in [10] which only covers errors; thus we name it synonym ambiguity.

5.2.3 Approach

The output of our API recognition, as covered in the Chapter 4, is a CRF-based classifier that is able to classify a token in a given Stack Overflow sentence as an API or not an API. After we have this CRF-based classifier, our task in this section is to map the recognized APIs to their formal forms written in the official documentation, i.e., API linking.

Figure 5.5 shows the main steps of our approach for API linking. The input is the content of the whole knowledge unit with APIs recognized. We perform the same text preprocessing to the input content as what we did in Section 4.2.3.1 of Chapter 4, after which it is pipelined through the three components, mention-mention similarity, scope filter, and mention-entry similarity. Mention-mention similarity measures the similarity between recognized API mentions. Scope filter comprises of a few Stack Overflow specific heuristics to help shrink the pool of linking candidates. And mention-entry similarity computes the similarity between an API mention and an entry in the knowledge base. These components sequentially eliminate the number of linking candidates to the recognized API. There can be three potential outputs: 1) No linking candidate found: there is no matching API in the knowledge base to the recognized API. This might happen due to a wrong recognition, or due to name variations or acronyms that we failed to normalize to match an entry in our knowledge base. 2) Exactly 1 linking candidate: this is the desired result since the API can be uniquely linked. 3) Multiple

FIGURE 5.5: Approach Overview: API linking

linking candidates: we will output a ranked list of APIs in this case. The first API on the list is evaluated as the linking output.

In the following two subsections, we describe the knowledge base population and the three components in Figure 5.5 in detail.

5.2.3.1 Knowledge Base Population

As mentioned, we need a knowledge base for API linking. Unlike existing entity linking work that uses Wikipedia as the knowledge base, there has been no knowledge base available for our API linking task. Therefore, we have to construct a knowledge base consisting the information of our studied Python APIs.

TABLE 5.8: Data Source for Knowledge Base Population

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

Each entry (or record) in our knowledge base contains 7 fields:

1. ID, i.e., the unique ID of an entry.
2. name. We store the fully qualified API as appeared on the official API documentation in this field.
3. URL, which is the unique URL address for the API.
4. library, which is the library the API belongs to (Pandas, Numpy, Matplotlib or Python Standard Library).

5. *class*, which represents the declaring class of the API if applicable. If the API is a method or function, then there has to be a class or a module hosting the API according to the naming convention of Python. If the API itself is already a class or module, then this field is populated with ‘N/A’.
6. *type*, e.g., if the API is a method or a class. In our case, we consider the API type of a Python module or class as *class*, and we consider the type of a Python function and method as *method*.
7. *description*. This field consists of the textual descriptions of the API from the official API documentation.

We crawl all these information for each library from their official site shown in Table 5.8. In total, our knowledge base is populated with 13713 records, in which 917 records are from Pandas, 2595 records are from Numpy, 4085 records are from Matplotlib, and 6116 records are from Python 2.7 standard library.

5.2.3.2 Linking to Knowledge Base Entries

Once we recognize an API mention, we match it against the “name” field of our knowledge base. Note that the “name” field stores fully qualified API name, while the API mentions on Stack Overflow are mostly unqualified. Hence, we design regular expressions to perform partial matching. We add a dot as the prefix if the mention is the simple name of the API. For example, if the recognized API mention is written as “apply”, our regular expression for partial matching will be “*.apply”. We also perform lightweight normalization to API mentions that are written in derivational forms, using the name variations observed when we constructed the API inventory (see Section 4.2.3.2 of Chapter 4). For example, “df.apply” will be normalized as “dataframe.apply” before we match it to the knowledge base entries. The matching to the knowledge base can potentially give us a list of linking candidates. We then use the three components we design i.e., mention-mention similarity, scope filter, and mention-entry similarity, to determine which linking candidate is the correct one to be linked to.

Mention-mention similarity. This component checks if there exists similar API mentions in the context that have been linked or is relatively unambiguous. For a given recognized API mention, we examine the global context within the knowledge unit by checking: 1) if the same API mention has been manually added with a URL link by Stack Overflow users; and 2) if a relatively formal version of the same API mention can be found. For example, in this knowledge unit <http://stackoverflow.com/questions/35782929>, “groupby” is an Pandas API. One of the “groupby” mentions has been added with a URL link to the official webpage of “groupby”. In this case, if some other “groupby” mentions without manual links are recognized as APIs by our API recognition, we link these “groupby” mentions to the same URL.

Scope filter. This component consists of a set of heuristics leveraging the global context of the current knowledge unit to narrow down the scope of API linking.

We first check if the declaring library of possible candidate can be found in the tags of the question or title of the question. This is done through simple lowercase string match. Stack Overflow tags help to sort questions into specific, well-defined categories so that we can have general idea about what the question is asking about. One Stack Overflow question has at least one tag. Similarly, the question title summarizes the topic of the question. With the title and tags information, we can potentially eliminate non-relevant candidates from other libraries.

Although we are performing API linking in text rather than the code blocks on Stack Overflow, we utilize the code blocks posted in the question or answer to help shrink the pool of linking candidate. Particularly, we use code blocks to find the potential declaring class (type) of the API mentioned in the nearby text, so as to resolve ambiguities for method or function names. Stack Overflow uses the Google Code Prettify Library¹ to perform lightweight syntax highlighting of code blocks in posts: code elements like keywords, types (classes), literal value, comments, etc., are augmented with `<span>` HTML tags of different classes. This provides a good source of information to design our scope filters, because the potential declaring classes can be obtained easily by selecting span elements whose class attribute is *typ*. Extracting the type information from the code blocks can be useful to cases where the post contains few textual descriptions.

¹<https://github.com/google/code-prettify>

We assume that a method is likely to be discussed together with its declaring class rather than other classes. Therefore, once a class can be found from the code blocks, we can use the class to help disambiguate method names recognized in the knowledge unit. We further use the recognized API mentions that are classes in the text of the knowledge unit (i.e., the recognized API is a class) for a similar job. Remember that in our knowledge base, for each API, we stored its type either as a class or a method. After we recognize an API mention, we further check its type by querying the knowledge base. If the type is class, we use it to disambiguate method mentions that are written in their simple names.

Mention-entry similarity. Although the scope filter can potentially reduce the number of potential candidates, it may not be able to reduce the number of linking candidates to only 1, i.e., finding the specific fully qualified API name, because 1) the tags and title matching only works at declaring library level; 2) we may not always find type/class from code blocks. Similarly, the mention-mention similarity component is not applicable in cases where there is no related reference URL or formal version of API mentions found in the discussion. To overcome these shortcomings, we compute the mention-entry similarity, which is the textual similarity between a knowledge unit and the documentation of an entry of the knowledge base. Specifically, after we apply the scope filter, we compare the content of the current knowledge unit with the content of each of the linking candidates. We represent the content of the knowledge unit and the content of the documentation of the entry as term frequency–inverse document frequency (TF-IDF) vectors. In our context, TF is the number of times a token appears in the current document while IDF is the inverse of the number of documents a token appears across the collection of documents. Here the collection of documents refers to the current knowledge unit plus all the descriptions of the linking candidates (one description is one document). We use logarithmic scaling because it is the standard way to reduce the distorting effect of code elements that are redundantly repeated in a single document [76]. We then calculate the textual similarity between the TF-IDF vector and each of the TF-IDF vectors of the linking candidates using cosine similarity, based on which we rank the linking candidates.

The three components are executed in sequence. The mention-mention similarity

component is determinative to the output once it is activated. The scope filter is to help reduce the number of linking candidates. And the mention-entry similarity component is used to rank the linking candidates.

5.2.4 Evaluation

In this section, we evaluate the performance of our API linking method.

Experimental Setup and Preliminary Results. We select recognized API mentions from Stack Overflow knowledge units that are tagged with either Pandas, Numpy or Matplotlib. Note these knowledge units may contain more than 1 tag. To evaluate the linking efficiency, all the API mentions chosen are method names, because method names generally have much more number of mappings in the knowledge base compared to class names, except for cases where a method is written in a relatively formal form, such as “dataframe.apply”, which only has one mapping in the knowledge base when we perform partial matching. In total, we select 120 API mentions, 60 of them are from Pandas questions, 30 from Matplotlib questions and 30 from Numpy questions. On average, each of the selected API mentions has 5.8 linking candidates. We create the ground truth data by manually inspecting the context of each API mention we selected to identify its correct linking result. We then compare the linking results given by our API linking approach with our ground truth data labelings.

The experimental results are shown in Table 5.9. We mark it as a true positive (TP) when our linked result is the same as the ground truth labeling. It is a false positive (FP) when the linked result does not match the labeling. A false negative (FN) is for the case when there is no linking result returned to the API mention but we would have expected to see a linking results. Precision is calculated as $TP/(TP+FP)$, recall is calculated as $TP/(TP+FN)$, and F1 is the harmonic mean of precision and recall. We can see that our API linking approach is able to achieve an overall F1-score of 0.884. Using our approach, the highest F1-score happens for the case of the Pandas Library. And we observe similar amount of FP and FN cases for the 3 studied libraries.

Discussion. The threats to validity in our evaluations involve: 1) the limited number of linking examples examined. In this section, we performed a preliminary evaluation

TABLE 5.9: Experimental Results of Our API Linking Approach

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

to our API linking approach by manually labeling a ground truth dataset consisting of 120 APIs for linking. The size of the ground truth dataset is smaller than the one in [77] (204 APIs). We will inspect more linking examples in the future. 2) our knowledge base size is small, only consisting of 13713 records crawled from 4 libraries. However, we believe the knowledge base size will not have a big impact on the linking performance of our approach, as we can always limit the scope for linking to only a few libraries or classes using our scope filter component.

In our experiments, we find that TF-IDF based cosine similarity does not always produce the highest similarity score for the correct API description. Sometimes it can only be useful to shrink the pool of candidates instead of picking the correct candidate directly like the mention-mention similarity component, which is similar to the scope filter component. For example, in this knowledge unit <http://stackoverflow.com/questions/32350288/>, for the linking of the recognized API *errorbar*, the mention-mention similarity component is not activated, and the scope filter is only able to shrink the linking scope to APIs of the Matplotlib Library, our approach links it to the wrong entry in the knowledge base (*matplotlib.axes.Axes.errorbar*) instead of the correct one (*matplotlib.pyplot.errorbar*) due to similar TF-IDF scores. In the future, we will consider the combination of multiple word representations, e.g., TF-IDF, topic distribution vector, word embeddings, for computing the mention-entry similarity.

5.2.5 Summary

In this section, we propose an API linking approach to link a recognized API mention in Stack Overflow discussions to its formal form in the official API documentation. The challenges lie in the inherent ambiguities of API mentions in informal natural language

text. We devise a set of heuristics leveraging the global context of a Stack Overflow knowledge unit to resolve the ambiguities. Our preliminary experiments show that the approach can disambiguate a recognized API mention with high precision and recall. While the linking between semantically linkable knowledge units studied in Section 5.1 of this chapter enriches the internal connections inside Stack Overflow, the API linking performed in this section bridges the knowledge in Stack Overflow externally to online API documentations. In the future, we will design more features to enhance our current heuristic-based API linking approach. We will also consider joint API recognition and linking techniques to reduce the error propagated from incorrect API recognitions. These future work will be discussed further in Chapter 6.

Chapter 6

Conclusions and Future Work

6.1 Conclusions

The software engineering community has not benefited from the advances of semantics-oriented applications like knowledge graph, which have greatly facilitated users' information seeking on the Web in general domains. To have such advanced applications in software engineering, we must be able to organize the online software engineering knowledge centered around software-specific entities and to build up the semantic linkages between different pieces of knowledge. Existing research from the software engineering community has overlooked these aspects. Existing research from the NLP and IR communities is not applicable to handle the unique characteristics brought by the socio-technical nature of software engineering social content.

With the broad goal of enabling domain-specific knowledge graph and semantic search in software engineering, in this thesis, we investigate the domain knowledge organization and extraction aspects in software engineering to move towards this goal. Using the software engineering social content on Stack Overflow, we identify a series of key research problems, including the understanding of online knowledge organization and connection in software engineering, the extraction of software-specific entities, and the auto-connection of linkable software knowledge.

First and foremost, we conduct an empirical study to understand the knowledge organization and connection on Stack Overflow by analyzing the knowledge network formed by Stack Overflow users' URL sharing activities. On Stack Overflow, due to the cross-reference of questions and answers (URL sharing), different knowledge units are connected forming a knowledge network. We develop a semi-automated methodology for qualitatively analyzing users' purposes for sharing URLs and quantitatively exposing the structure and dynamics of the knowledge network. Our study shows that knowledge units on Stack Overflow can be linkable for various purposes. The structure of the knowledge network in terms of the indegree distribution follows a power-law distribution (i.e., stable) and evolves to stability quickly, in spite of the ad-hoc and opportunistic nature of URL sharing activities, which implies that a large amount of knowledge units in the long tail of the power law distribution are under linked. Furthermore, the study shows the knowledge network is created by the crowdsourced power of its users, while it can be time consuming to purely rely on active users to digest and then integrate the large amount of information generated by millions of users into the knowledge network. These observations from our study call for advanced ways to manage the online knowledges techniques, e.g., organizing the knowledge based on software entities, and to automatically connect linkable software engineering knowledge.

A fundamental step to organize software engineering social content centered around entities is to extract the software-specific entities. In this thesis, we formulate the research problem of NER in software engineering. We have exposed the key decisions to be made for handling the design challenges brought by unique characteristics of domain-specific texts on Stack Overflow. We build S-NER, a semi-supervised 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, and a rich and effective set of features for NER in software engineering texts. We further propose an improved method for API recognition, which address two fundamental challenges in recognizing API mentions from informal natural language text, i.e., the polysemy issue

and the sentence-format variations issue. Polysemy refers to the ambiguous sense of an API name, e.g., Python API names, such as `apply` and `append`, can also be common words. Sentence-format variations refer to the lack of consistent sentence writing format for inferring API mentions. We tackle the challenges by exploiting name synonyms and rich semantic context of API mentions. We use Brown Clusters and clusters of word embeddings learned from the abundant unlabeled text as features to model sentence-format variations. We also utilize self-training to further compensate for the lack of labeled training data. We apply our approach on recognizing API mentions in Stack Overflow’s informal textual discussions that suffer from the polysemy and sentence-format variation issues to demonstrate its effectiveness.

We have developed automated techniques to connect linkable software engineering knowledge discussed on Stack Overflow. At document level, we show that knowledge units often contain semantically relevant knowledge, and they can be linkable for different purposes. we formulate the problem of predicting semantically linkable knowledge units as a multiclass classification problem, and solve the problem using deep learning techniques. To overcome the issue that linkable knowledge units may share few words in common, we feed word embeddings to convolutional neural network (CNN) to capture the word-level and document-level semantics of knowledge units. Instead of using human-engineered classifier features which are hard to design for informal user-generated content, we exploit large amounts of different types of user-created knowledge-unit links to train the CNN to learn the most informative features for the multiclass classification task. Our evaluation shows that our deep learning based approach significantly and consistently outperforms existing methods using traditional word representations and human-engineered features. At entity level, we link an recognized API mention in the textual discussions to its full qualified name, known as fine-grained API linking, based on our API recognition results. To deal with the inherent ambiguities of the API mentions in informal social text, we have devised a pipeline of Stack Overflow specific heuristics to disambiguate an recognized API mention to its formal form appeared in the official API documentation. Our API linking approach consists of three components, i.e., mention-mention similarity, scope filter, and mention-entry similarity. We have shown that our approach is able to deliver high precision and recall

when linking API mentions on Stack Overflow.

We expect this thesis to serve as a starting point for further research in moving towards domain-specific knowledge graph and semantic search in software engineering.

6.2 Future Work

While we have performed an initial work towards knowledge graph and semantic search in software engineering, a lot of work remains to be done. We briefly discuss a list of future work that can be built upon what we have done in this thesis.

6.2.1 Mining Other Types of Software Artifacts

All the techniques developed in this thesis are based on software engineering social content, particularly the informal textual discussions on Stack Overflow. However, the ideas we propose are not limited to mining the textual content on Stack Overflow. They can be easily adapted to other types of software artifacts as well, such as developer emails and bug reports. One potential direction is to apply our semantic relatedness detection method developed in Section 5.1 of Chapter 5 to duplicate bug reports detection. Similar to the crowdsourced nature of knowledge units on Stack Overflow, bug reporting is also an uncoordinated distributed process [14]. Software testers may report the same bug more than once in the bug reporting system. Moreover, bug reports are written manually by developers based on their understanding of the observed bug. Different developers possess different levels of knowledge and have their own writing styles. Even for the same bug, different developers may describe it using words and sentences that have few literal overlaps, which is again quite similar to the case of semantically equivalent knowledge units on Stack Overflow. The auto-detection of duplicate bug reports can help reduce bug checker's efforts in identifying repeated bug reports, and save developers' time in fixing the same bug. Existing approaches on detecting duplicate bug supply information retrieval based features, such as TF-IDF representation of words, to classification models. For example, in [15], Sun et al. train a Support Vector Machine

(SVM) classifier with TF-IDF as the features to detect duplicate bug reports. An information retrieval based similarity measure (BM25F [115]) is used in [14]. Nguyen et al. [12] combine the information retrieval based *BM25F* used in [14] with topic modeling (LDA) to further improve the detection accuracy of duplicate bug reports. However, as we have pointed out in Section 5.1 of Chapter 5, the hand-crafted information retrieval based features used in these work do not capture the word-level and sentence-level semantics well. To overcome this shortcoming, we can migrate our convolutional neural network (CNN) and word embedding based approach to duplicate bug reports detection. We can first collect a large amount of bug reports to train word embeddings. The input unit to CNN is a pair of bug reports that have been marked as duplicate, the loss function of CNN can still be the cosine similarity. To evaluate the CNN plus word embedding approach, similar to our work in Section 5.1 of Chapter 5, we can first prepare a set of duplicate bug reports in pairs, and then apply the trained model on a pair of duplicates.

6.2.2 Towards Guided Search with Knowledge Location Map

As we have discussed in Section 3.9.2 of Chapter 3, when browsing Stack Overflow, it is very likely that a user lands on a knowledge unit that is relevant but does not fully satisfy his information needs. In this case, the navigational support for developers (especially novices) to guide the information seeking would be useful. The key idea, as discussed, is to make hidden knowledge clusters, their hub and betweenness unit, their topics and their correlations easily accessible in an *interactive knowledge location map*. Based on our findings in Chapter 3, the knowledge network of Stack Overflow can be well partitioned into knowledge clusters, and the knowledge units within a knowledge cluster are semantically similar. While the user is navigating the 1-dimension list of question and answers on the Stack Overflow webpage of a knowledge unit, we believe that providing a 2-dimension visualization of the sub knowledge network alongside the webpage, showing their connections, highlighting the knowledge unit the user is in, summarizing the topics of each cluster in the sub knowledge network and highlighting important software-specific entities in the current knowledge unit itself using the entity extraction techniques we developed in Chapter 4, can quickly inform the user “where he

is now and where he can go”. This is to mimic the directory (location map) of shopping malls. Still, the user is free to choose his way to search and re-search for the desired information. The user is also free to not navigate through these non-isolated knowledge network. The location map will not impede the user’s normal information seeking.

As per the useful scenarios of the proposed *2D knowledge location map*, we admit it is not going to help with the navigability and query formulation in every user case. Certainly there are many situations in which some queries can lead to the desired results quickly without much navigation of the knowledge network. However, interestingly, our study indicates that the community feedback of non-isolated knowledge units is significantly better than that of the isolated knowledge units. This implies that non-isolated knowledge units are more popular and have attracted much more user attention compared with isolated knowledge units. That is to say, the non-isolated knowledge units can potentially receive much more visits. The good community feedback of non-isolated units, together with their highly correlated nature, is telling that the knowledge density of a non-isolated sub knowledge network is generally larger than an isolated one. This can potentially introduce navigation difficulties for novice developers. Considering the fact that one quarter of the total knowledge units are connected, which is already a non-trivial proportion, plus the fact that more user attention is on the non-isolated knowledge units, we believe providing a knowledge location map is necessary for guiding the exploratory search within Stack Overflow. The 2D visualization of the location map, together with the topics of each knowledge cluster, will provide an additional channel for the user to navigate the knowledge base of Stack Overflow; the topic summarization, together with some highlighting on the software-specific entities discussed in the knowledge units, can potentially help with user’s query reformulation.

Furthermore, we could also incorporate the semantically linkable knowledge prediction technique we proposed in Section 5.1 of Chapter 5 to the interactive knowledge location map. The knowledge location map discussed above is based on the connected knowledge units in the knowledge network of Stack Overflow. As we pointed out in Section 5.1, a large proportion of knowledge units in Stack Overflow are unlinked, while many of the unlinked knowledge units can be potentially useful. Therefore, we

can further deploy a component in the location map to recommend to users the unlinked semantically related knowledge.

6.2.3 Improving Fine-grained API Linking

Learning to link APIs. As a preliminary study on fine-grained API linking from natural language text in Stack Overflow, our proposed API linking method in Section 5.2 in Chapter 5 is based on a limited set of heuristics (or features) that are specific to Stack Overflow. There are several limitations of our current API linking approach, and we can improve it in various aspects. First, the mention-mention similarity component checks the textual content in the same knowledge unit if there exists build-in URLs for the same API mention. This component is essentially rule-based and will not be activated when there are no build-in URLs appeared in the current knowledge unit. To overcome this issue, we can check the whole Stack Overflow knowledge base to examine the presence of a manually added URL on the same API mention or on its informal derivative forms. The contextual similarity between the API mention with a URL and the current API to be linked can be utilized to assist the disambiguation. The contextual similarity can be computed at different levels: 1) the similarity between two knowledge units; 2) the similarity between two posts (a post refers either a question body, an answer body, or a comment); 3) the similarity between two sentences containing API mentions. To ensure unbiased results, we can adopt different word representations for computing similarity, e.g., TF-IDF, topic distribution vectors, word embeddings, etc. We can further add other features for estimating mention-mention similarity, e.g., whether two posts are published by the same Stack Overflow user. Second, as we have discussed in Section 5.2.4, using TF-IDF for computing mention-entry similarity leads to unsatisfiable ranking of linking candidates. As studied by Sun et al. [15], excluding TF when computing similarities between two software artifacts delivers better results than using TF-IDF. We will try out IDF and other word representations in the future for computing mention-entry similarity. We will also consider the edit distance similarity between the spelling of the API mention and the spellings of formal APIs in the knowledge base, so as to detect whether a API mention is an abbreviated form. Third, these above-mentioned

features (or heuristics) we are proposing are not of equal importance. We can resort to machine learning to automatically determine the importances of these features. We can combine these features in a linear model and assign each feature with a weight. The training and decoding processes will be similar to the entity linking work done by Liu et al. [120]. Such machine learning based approach would be more robust and tolerant to the contextual variations of API mentions among knowledge units.

Joint Recognition and Linking of API Mentions. In our work, the recognition and linking of API mentions are conducted separately, and there is no feedback from the API linking to the API recognition. As a result, the error of API recognition can be propagated to API linking, leading to false negative cases, as evidenced in Section 5.2 of Chapter 5. A possible direction to overcome this issue is to perform joint API recognition and linking, such that recognition part and the linking part can complement to each other. This is inspired from recent work on joint entity recognition and linking [152, 153], especially [153], which jointly recognize and link location mentions (location is an entity type) in tweets. They only have one entity type, i.e., location, hence, the consistency issues between entity type and category considered in [152] becomes less meaningful. In our case, we also only have one entity type to be recognized and linked, i.e., API. Also, the genre of text we focus on, i.e., Stack Overflow, is also informal, similar to the social text on twitter. Such a joint approach in [153], when adapted properly to our scenario, would improve the performance of API recognition, and API linking in Stack Overflow could benefit naturally as it requires the correct recognition of APIs. For example, in <http://stackoverflow.com/q/26635870/>, for the sentence “You can change the `reduce.C.function` option”, our current API recognition proposed in Chapter 4 mistakenly recognize the token “`reduce.C.function`” as an API. However, this API recognition has no linkable entries in our knowledge base. This feedback information can be utilized to inform our API recognizer to avoid such mistakes.

6.2.4 Entity Relation Extraction in Software Engineering

Software-specific entities mentioned in software artifacts are not standalone and have relations to each other. For example, in the sentence “you can use the `apply` method

declared in the dataframe class of the Pandas Library”, there are three software-specific named entities mentioned, i.e., *apply*, *dataframe*, and *Pandas*, with entity type *API*, *API*, and *tool-library-framework*, respectively. We can infer from the sentence that the entity *apply* is a sub-method of the entity *dataframe*, and both these two entities are subsidiaries to the entity *Pandas*. While we have conducted a study in extracting entity mentions in this thesis, the extraction of relations between domain-specific entities in software engineering has not been touched.

Relation extraction plays an important role in extracting structured information from unstructured text [154], which is another important step in moving towards knowledge graphs and semantic search in software engineering. Similar to the relation extraction work on general entities, e.g., relationships between different people [155], we can build a searchable knowledge base hosting the relationships among software-specific entities. To this end, we have to first define certain types of relation between software-specific entities, just like we have to define the entity categories before we perform named entity recognition in Section 4.1 of Chapter 4. We can get some inspirations on how to define the relations between entities from the relation types defined at the ACE 2003 community task: <http://www.itl.nist.gov/iad/mig/tests/ace/2003/>. A starting point would be to define the relationships among different APIs. Some example relationship types include: API *A* is the parent of API *B* (i.e., *B* is directly defined under *A*), API *B* is a sibling to API *C*, etc. We then have to label training sentences that contain the entity relationship types we define. With these labeled sentences, we can proceed to extract textual features, which is similar the feature extraction done in Chapter 4. Some example features can be the pair of entity types in the relationship, bag-of-words features, syntactic features, etc. After we have enough training data with textual features extracted, the entity relation extraction problem can be transformed into a classification problem. We can train a classification model for every relation type we define. We will explore this direction in the future.

Appendix A

List of Publications

- Deheng Ye, and Zhenchang Xing. API Recognition and Linking in Developer Online Forums. *Ongoing work* ¹.
- Deheng Ye*, Bowen Xu*, Zhenchang Xing, Xin Xia, Guibin Chen, and Shanping Li. Predicting Semantically Linkable Knowledge in Developer Online Forums via Convolutional Neural Network. The 31st IEEE/ACM International Conference on Automated Software Engineering (ASE 2016). (* Joint first authors)
- Deheng Ye, Zhenchang Xing, Chee Yong Foo, Jing Li, and Nachiket Kapre. Learning to Extract API Mentions from Informal Natural Language Discussions. The 32nd International Conference on Software Maintenance and Evolution (IC-SME 2016).
- Deheng Ye, Zhenchang Xing, and Nachiket Kapre. The Structure and Dynamics of Knowledge Network in Domain-specific Q&A Sites: A Case Study of Stack Overflow. Journal of Empirical Software Engineering (EMSE). 2016. (IF: 2.161)
- Deheng Ye, Zhenchang Xing, Chee Yong Foo, Zi Qun Ang, Jing Li, and Nachiket Kapre. Software-specific Named Entity Recognition in Software Engineering Social Content. The 23rd IEEE International Conference on Software Analysis, Evolution, and Reengineering (SANER 2016).

¹By the time when this thesis was submitted.

- Deheng Ye, Zhenchang Xing, Jing Li, and Nachiket Kapre. Software-specific Part-of-speech Tagging: An Experimental Study on Stack Overflow. The 31st ACM/SIGAPP Symposium on Applied Computing (SAC 2016).
- Jing Li, Zhenchang Xing, Deheng Ye, and Xuejiao Zhao. From Discussion to Wisdom: Web Resource Recommendation for Hyperlinks in Stack Overflow. The 31st ACM/SIGAPP Symposium on Applied Computing (SAC 2016).
- Lingfeng Bao, Deheng Ye, Zhenchang Xing, Xin Xia, and Xinyu Wang. ActivitySpace: A Remembrance Framework to Support Interapplication Information Needs. The 30th IEEE/ACM International Conference on Automated Software Engineering (ASE 2015).
- Nachiket Kapre, and Deheng Ye. GPU-Accelerated High-Level Synthesis for Bitwidth Optimization of FPGA Datapaths. The 24th ACM/SIGDA International Symposium on Field-Programmable Gate Arrays (FPGA 2016).
- Deheng Ye, and Nachiket Kapre. MixFX-SCORE: Heterogeneous Fixed-Point Compilation of Dataflow Computations. The 22nd IEEE International Symposium on Field-Programmable Custom Computing Machines (FCCM 2014).

Bibliography

- [1] Lena Mamykina, Bella Manoim, Manas Mittal, George Hripcsak, and Björn Hartmann. Design lessons from the fastest q&a site in the west. In *Proceedings of the SIGCHI conference on Human factors in computing systems*, pages 2857–2866. ACM, 2011.
- [2] Chris Parnin, Christoph Treude, Lars Grammel, and Margaret-Anne Storey. Crowd documentation: Exploring the coverage and the dynamics of api discussions on stack overflow. *Georgia Institute of Technology, Tech. Rep*, 2012.
- [3] Edgar Meij, Krisztian Balog, and Daan Odijk. Entity linking and retrieval for semantic search. In *WSDM*, pages 683–684, 2014.
- [4] Patrick Pantel and Ariel 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*, pages 83–92. Association for Computational Linguistics, 2011.
- [5] Erik F. Tjong Kim Sang and Fien De Meulder. Introduction to the conll-2003 shared task: Language-independent named entity recognition. In *Proceedings of the Seventh Conference on Natural Language Learning at HLT-NAACL 2003 - Volume 4*, CONLL '03, pages 142–147. Association for Computational Linguistics, 2003.
- [6] Thomas Lin, Patrick Pantel, Michael Gamon, Anitha Kannan, and Ariel Fuxman. Active objects: Actions for entity-centric search. In *Proceedings of the 21st international conference on World Wide Web*, pages 589–598. ACM, 2012.

- [7] Mark J Carman, Fabio Crestani, Morgan Harvey, and Mark Baillie. Towards query log based personalization using topic models. In *Proceedings of the 19th ACM international conference on Information and knowledge management*, pages 1849–1852. ACM, 2010.
- [8] Guangyou Zhou, Tingting He, Jun Zhao, and Po 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)*, pages 250–259. Association for Computational Linguistics Beijing, China, 2015.
- [9] Alberto Bacchelli, Michele Lanza, and Romain Robbes. Linking e-mails and source code artifacts. In *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering-Volume 1*, pages 375–384. ACM, 2010.
- [10] Barthélemy Dagenais and Martin P Robillard. Recovering traceability links between an api and its learning resources. In *Software Engineering (ICSE), 2012 34th International Conference on*, pages 47–57. IEEE, 2012.
- [11] Henrique Rocha, Marco Tulio Valente, Humberto Marques-Neto, and Gail C. Murphy. An empirical study on recommendations of similar bugs. In *SANER*, 2016.
- [12] Anh Tuan Nguyen, Tung Thanh Nguyen, Tuan N Nguyen, Daniel Lo, and Chengnian Sun. Duplicate bug report detection with a combination of information retrieval and topic modeling. In *Automated Software Engineering (ASE), 2012 Proceedings of the 27th IEEE/ACM International Conference on*, pages 70–79. IEEE, 2012.
- [13] Yuan Tian, Chengnian Sun, and David Lo. Improved duplicate bug report identification. In *Software Maintenance and Reengineering (CSMR), 2012 16th European Conference on*, pages 385–390. IEEE, 2012.
- [14] Chengnian Sun, David Lo, Siau-Cheng Khoo, and Jing Jiang. Towards more accurate retrieval of duplicate bug reports. In *Proceedings of the 2011 26th*

- IEEE/ACM International Conference on Automated Software Engineering*, pages 253–262. IEEE Computer Society, 2011.
- [15] Chengnian Sun, David Lo, Xiaoyin Wang, Jing Jiang, and Siau-Cheng Khoo. A discriminative model approach for accurate duplicate bug report retrieval. In *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering-Volume 1*, pages 45–54. ACM, 2010.
- [16] Yun Zhang, David Lo, Xin Xia, and Jian-Ling Sun. Multi-factor duplicate question detection in stack overflow. *Journal of Computer Science and Technology*, 30(5):981–997, 2015.
- [17] Yuan Tian and David Lo. A comparative study on the effectiveness of part-of-speech tagging techniques on bug reports. In *SANER*, pages 570–574, 2015.
- [18] Deheng Ye, Zhenchang Xing, Jing Li, and Nachiket Kapre. Software-specific part-of-speech tagging: An experimental study on stack overflow. In *Proceedings of the 31st Annual ACM Symposium on Applied Computing, SAC '16*, pages 1378–1385, New York, NY, USA, 2016. ACM.
- [19] Lev Ratinov and Dan Roth. Design challenges and misconceptions in named entity recognition. In *Proceedings of the Thirteenth Conference on Computational Natural Language Learning*, pages 147–155. Association for Computational Linguistics, 2009.
- [20] Einat Minkov, Richard C Wang, and William 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*, pages 443–450. Association for Computational Linguistics, 2005.
- [21] Xiaohua Liu, Shaodian Zhang, Furu Wei, and Ming Zhou. Recognizing named entities in tweets. In *Proceedings of the 49th Annual Meeting of the Association for Computational Linguistics: Human Language Technologies-Volume 1*, pages 359–367. Association for Computational Linguistics, 2011.

- [22] Alan Ritter, Sam Clark, Oren Etzioni, et al. Named entity recognition in tweets: an experimental study. In *Proceedings of the Conference on Empirical Methods in Natural Language Processing*, pages 1524–1534. Association for Computational Linguistics, 2011.
- [23] Jin-Dong Kim, Tomoko Ohta, Yoshimasa Tsuruoka, Yuka Tateisi, and Nigel 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, JNLPBA '04*, pages 70–75. Association for Computational Linguistics, 2004.
- [24] Kazuhiro Yoshida and Jun'ichi Tsujii. Reranking for biomedical named-entity recognition. In *Proceedings of the Workshop on BioNLP 2007: Biological, Translational, and Clinical Language Processing*, pages 209–216. Association for Computational Linguistics, 2007.
- [25] Yefeng Wang. Annotating and recognising named entities in clinical notes. In *Proceedings of the ACL-IJCNLP 2009 Student Research Workshop*, pages 18–26. Association for Computational Linguistics, 2009.
- [26] Yangjie Yao and Aixin Sun. Mobile phone name extraction from internet forums: a semi-supervised approach. *World Wide Web*, pages 1–23, 2015.
- [27] John D. Lafferty, Andrew McCallum, and Fernando C. N. Pereira. Conditional random fields: Probabilistic models for segmenting and labeling sequence data. In *Proceedings of the Eighteenth International Conference on Machine Learning, ICML '01*, pages 282–289, 2001. ISBN 1-55860-778-1.
- [28] Peter F Brown, Peter V Desouza, Robert L Mercer, Vincent J Della Pietra, and Jenifer C Lai. Class-based n-gram models of natural language. *Computational linguistics*, 18(4):467–479, 1992.
- [29] Percy Liang. *Semi-supervised learning for natural language*. PhD thesis, Cite-seer, 2005.

- [30] Tomas Mikolov, Ilya Sutskever, Kai Chen, Greg S Corrado, and Jeff Dean. Distributed representations of words and phrases and their compositionality. In *Advances in neural information processing systems*, pages 3111–3119, 2013.
- [31] Joseph Turian, Lev Ratinov, and Yoshua 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*, pages 384–394. Association for Computational Linguistics, 2010.
- [32] Dan Wu, Wee Sun Lee, Nan Ye, and Hai Leong 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*, pages 1523–1532. Association for Computational Linguistics, 2009.
- [33] Tomas Mikolov, Kai Chen, Greg Corrado, and Jeffrey Dean. Efficient estimation of word representations in vector space. *arXiv preprint arXiv:1301.3781*, 2013.
- [34] David H Hubel and Torsten N Wiesel. Receptive fields of single neurones in the cat’s striate cortex. *The Journal of physiology*, 148(3):574–591, 1959.
- [35] F Maxwell Harper, Daphne Raban, Sheizaf Rafaeli, and Joseph A Konstan. Predictors of answer quality in online q&a sites. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, pages 865–874. ACM, 2008.
- [36] Lada A Adamic, Jun Zhang, Eytan Bakshy, and Mark S Ackerman. Knowledge sharing and yahoo answers: everyone knows something. In *Proceedings of the 17th international conference on World Wide Web*, pages 665–674. ACM, 2008.
- [37] Miltiadis Allamanis and Charles Sutton. Why, when, and what: analyzing stack overflow questions by topic, type, and code. In *Proceedings of the 10th Working Conference on Mining Software Repositories*, pages 53–56. IEEE Press, 2013.
- [38] Anton Barua, Stephen W Thomas, and Ahmed E Hassan. What are developers talking about? an analysis of topics and trends in stack overflow. *Empirical Software Engineering*, pages 619–654, 2014.

- [39] Kartik Bajaj, Karthik Pattabiraman, and Ali Mesbah. Mining questions asked by web developers. In *Proceedings of the 11th Working Conference on Mining Software Repositories*, pages 112–121. ACM, 2014.
- [40] Christoffer Rosen and Emad Shihab. What are mobile developers asking about? a large scale study using stack overflow. *Empirical Software Engineering*, pages 1–32, 2015.
- [41] David M Blei, Andrew Y Ng, and Michael I Jordan. Latent dirichlet allocation. *the Journal of Machine Learning Research*, pages 993–1022, 2003.
- [42] Paul Fugelstad, Patrick Dwyer, Jennifer Filson Moses, John Kim, Cleila Anna Mannino, Loren Terveen, and Mark Snyder. What makes users rate (share, tag, edit...)?: predicting patterns of participation in online communities. In *Proceedings of the ACM 2012 conference on Computer Supported Cooperative Work*, pages 969–978. ACM, 2012.
- [43] Aditya Pal, Shuo Chang, and Joseph A Konstan. Evolution of experts in question answering communities. In *ICWSM*, 2012.
- [44] Aditya Pal, F. Maxwell Harper, and Joseph A. Konstan. Exploring question selection bias to identify experts and potential experts in community question answering. *ACM Trans. Inf. Syst.*, 30(2):10:1–10:28, May 2012. ISSN 1046-8188.
- [45] Shaowei Wang, Daniel Lo, Bogdan Vasilescu, and Alexander Serebrenik. Entagrec: an enhanced tag recommendation system for software information sites. In *Software Maintenance and Evolution (ICSME), 2014 IEEE International Conference on*, pages 291–300. IEEE, 2014.
- [46] Mohammad Masudur Rahman, Shamima Yeasmin, and Chanchal K Roy. Towards a context-aware ide-based meta search engine for recommendation about programming errors and exceptions. In *Software Maintenance, Reengineering and Reverse Engineering (CSMR-WCRE), 2014 Software Evolution Week-IEEE Conference on*, pages 194–203. IEEE, 2014.

- [47] Ashton Anderson, Daniel Huttenlocher, Jon Kleinberg, and Jure Leskovec. Discovering value from community activity on focused question answering sites: a case study of stack overflow. In *Proceedings of the 18th ACM SIGKDD international conference on Knowledge Discovery and Data Mining*, pages 850–858. ACM, 2012.
- [48] Bogdan Vasilescu, Alexander Serebrenik, Prem Devanbu, and Vladimir Filkov. How social q&a sites are changing knowledge sharing in open source software communities. In *Proceedings of the 17th ACM conference on Computer supported cooperative work & social computing*, pages 342–354. ACM, 2014.
- [49] Christoph Treude, Ohad Barzilay, and Margaret-Anne Storey. How do programmers ask and answer questions on the web?: Nier track. In *Software Engineering (ICSE), 2011 33rd International Conference on*, pages 804–807. IEEE, 2011.
- [50] L. Guerrouj, S. Azad, and P.C. Rigby. The influence of app churn on app success and stackoverflow discussions. In *Software Analysis, Evolution and Reengineering (SANER), 2015 IEEE 22nd International Conference on*, pages 321–330, March 2015.
- [51] Shaowei Wang, David Lo, and Lingxiao Jiang. An empirical study on developer interactions in stackoverflow. In *Proceedings of the 28th Annual ACM Symposium on Applied Computing*, pages 1019–1024. ACM, 2013.
- [52] Joshua Sunshine, James D Herbsleb, and Jonathan Aldrich. Searching the state space: A qualitative study of api protocol usability. In *Proceedings of the 22Nd International Conference on Program Comprehension, ICPC 2015*, 2015.
- [53] Megan Squire. Should we move to stack overflow?: Measuring the utility of social media for developer support. In *Proceedings of the 37th International Conference on Software Engineering - Volume 2, ICSE '15*, pages 219–228, Piscataway, NJ, USA, 2015. IEEE Press.
- [54] Yla R Tausczik, Aniket Kittur, and Robert E Kraut. Collaborative problem solving: a study of mathoverflow. In *Proceedings of the 17th ACM conference*

- on Computer supported cooperative work & social computing*, pages 355–367. ACM, 2014.
- [55] Nicole Novielli, Fabio Calefato, and Filippo Lanubile. The challenges of sentiment detection in the social programmer ecosystem. In *Proceedings of the 7th International Workshop on Social Software Engineering*, pages 33–40. ACM, 2015.
- [56] Carlos Gómez, Brendan Cleary, and Leif Singer. A study of innovation diffusion through link sharing on stack overflow. In *Mining Software Repositories (MSR), 2013 10th IEEE Working Conference on*, pages 81–84. IEEE, 2013.
- [57] Jun Zhang, Mark S Ackerman, and Lada Adamic. Expertise networks in online communities: structure and algorithms. In *Proceedings of the 16th international conference on World Wide Web*, pages 221–230. ACM, 2007.
- [58] Akshay Java, Xiaodan Song, Tim Finin, and Belle Tseng. Why we twitter: understanding microblogging usage and communities. In *Proceedings of the 9th WebKDD and 1st SNA-KDD 2007 workshop on Web mining and social network analysis*, pages 56–65. ACM, 2007.
- [59] Harry Halpin, Valentin Robu, and Hana Shepherd. The complex dynamics of collaborative tagging. In *Proceedings of the 16th international conference on World Wide Web*, pages 211–220. ACM, 2007.
- [60] Silviu Cucerzan. Large-scale named entity disambiguation based on wikipedia data. In *EMNLP-CoNLL*, volume 7, pages 708–716, 2007.
- [61] Andrei Broder, Ravi Kumar, Farzin Maghoul, Prabhakar Raghavan, Sridhar Rajagopalan, Raymie Stata, Andrew Tomkins, and Janet Wiener. Graph structure in the web. *Computer networks*, 33(1):309–320, 2000.
- [62] Aaron Clauset, Cosma Rohilla Shalizi, and Mark EJ Newman. Power-law distributions in empirical data. *SIAM review*, 51(4):661–703, 2009.
- [63] Robert Meusel, Sebastiano Vigna, Oliver Lehmborg, and Christian Bizer. Graph structure in the web—revisited: a trick of the heavy tail. In *Proceedings of the*

- companion publication of the 23rd international conference on World wide web companion*, pages 427–432. International World Wide Web Conferences Steering Committee, 2014.
- [64] Scott A Golder and Bernardo A Huberman. Usage patterns of collaborative tagging systems. *Journal of information science*, 32(2):198–208, 2006.
- [65] Ciro Cattuto, Vittorio Loreto, and Luciano Pietronero. Semiotic dynamics and collaborative tagging. *Proceedings of the National Academy of Sciences*, 104(5):1461–1464, 2007.
- [66] Wai-Tat Fu, Thomas Kannampallil, Ruogu Kang, and Jibo He. Semantic imitation in social tagging. *ACM Transactions on Computer-Human Interaction (TOCHI)*, 17(3):12, 2010.
- [67] Albert-László Barabási and Réka Albert. Emergence of scaling in random networks. *Science*, 286(5439):509–512, 1999.
- [68] A.E. Hassan and R.C. Holt. The small world of software reverse engineering. In *Reverse Engineering, 2004. Proceedings. 11th Working Conference on*, pages 278–283, 2004.
- [69] Roger Pressman. *Software Engineering: A Practitioner’s Approach*. McGraw-Hill, Inc., New York, NY, USA, 7 edition, 2010.
- [70] Panagiotis Louridas, Diomidis Spinellis, and Vasileios Vlachos. Power laws in software. *ACM Trans. Softw. Eng. Methodol.*, 18(1):2:1–2:26, October 2008. ISSN 1049-331X.
- [71] Chenliang Li and Aixin 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*, pages 43–52. ACM, 2014.
- [72] Wenhui Liao and Sriharsha 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*, pages 58–65. Association for Computational Linguistics, 2009.
- [73] Rada Mihalcea. Co-training and self-training for word sense disambiguation. In *CoNLL*, pages 33–40, 2004.
- [74] Roberto Navigli. Word sense disambiguation: A survey. *ACM Computing Surveys (CSUR)*, 41(2):10, 2009.
- [75] Xinxiong Chen, Zhiyuan Liu, and Maosong Sun. A unified model for word sense representation and disambiguation. In *EMNLP*, pages 1025–1035. Citeseer, 2014.
- [76] Peter C Rigby and Martin P Robillard. Discovering essential code elements in informal documentation. In *Proceedings of the 2013 International Conference on Software Engineering*, pages 832–841. IEEE Press, 2013.
- [77] Siddharth Subramanian, Laura Inozemtseva, and Reid Holmes. Live api documentation. In *Proceedings of the 36th International Conference on Software Engineering*, pages 643–652. ACM, 2014.
- [78] Giuliano Antoniol, Gerardo Canfora, Gerardo Casazza, Andrea De Lucia, and Ettore Merlo. Recovering traceability links between code and documentation. *Software Engineering, IEEE Transactions on*, 28(10):970–983, 2002.
- [79] Barthélemy Dagenais and Harold Ossher. Automatically locating framework extension examples. In *Proceedings of the 16th ACM SIGSOFT International Symposium on Foundations of software engineering*, pages 203–213. ACM, 2008.
- [80] Andrian Marcus, Jonathan Maletic, et al. Recovering documentation-to-source-code traceability links using latent semantic indexing. In *Software Engineering, 2003. Proceedings. 25th International Conference on*, pages 125–135. IEEE, 2003.
- [81] Andrea De Lucia, Rocco Oliveto, and Genoveffa Tortora. Adams re-trace: traceability link recovery via latent semantic indexing. In *Proceedings of the 30th international conference on Software engineering*, pages 839–842. ACM, 2008.

- [82] J-R Falleri, Marianne Huchard, Mathieu Lafourcade, Clementine Nebut, Violaine Prince, and Michel Dao. Automatic extraction of a wordnet-like identifier network from software. In *Program Comprehension (ICPC), 2010 IEEE 18th International Conference on*, pages 4–13. IEEE, 2010.
- [83] Swastik Gupta, Sana Malik, Lori 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*, pages 3–12. IEEE, 2013.
- [84] Hsin-Yi Jiang, Tien N Nguyen, Xiang Chen, Hojun Jaygarl, and Carl 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*, pages 59–68. IEEE Computer Society, 2008.
- [85] Alberto Bacchelli, Marco D’Ambros, Michele Lanza, and Romain Robbes. Benchmarking lightweight techniques to link e-mails and source code. In *Reverse Engineering, 2009. WCRE’09. 16th Working Conference on*, pages 205–214. IEEE, 2009.
- [86] Mohammad Masudur Rahman, Chanchal K. Roy, and David Lo. Rack: Automatic api recommendation using crowdsourced knowledge. In *SANER*, 2016.
- [87] Wujie Zheng, Qirun Zhang, and Michael 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*, pages 480–483. ACM, 2011.
- [88] Fuxiang Chen and Sunghun Kim. Crowd debugging. In *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*, pages 320–332. ACM, 2015.
- [89] Qing Gao, Hansheng Zhang, Jie Wang, Yingfei Xiong, Lu Zhang, and Hong Mei. Fixing recurring crash bugs via analyzing q&a sites (t). In *Automated Software*

- Engineering (ASE), 2015 30th IEEE/ACM International Conference on*, pages 307–318. IEEE, 2015.
- [90] Leon Moonen. Generating robust parsers using island grammars. In *Reverse Engineering, 2001. Proceedings. Eighth Working Conference on*, pages 13–22. IEEE, 2001.
- [91] Alberto Bacchelli, Anthony Cleve, Michele Lanza, and Andrea Mocci. Extracting structured data from natural language documents with island parsing. In *Automated Software Engineering (ASE), 2011 26th IEEE/ACM International Conference on*, pages 476–479. IEEE, 2011.
- [92] Giovanni Capobianco, Andrea De Lucia, Rocco Oliveto, Annibale Panichella, and Sebastiano Panichella. Improving ir-based traceability recovery via noun-based indexing of software artifacts. *Journal of Software: Evolution and Process*, 25(7):743–762, 2013.
- [93] Ramin Shokripour, John Anvik, Zarinah M Kasirun, and Sima 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*, pages 2–11. IEEE Press, 2013.
- [94] Claudia Hauff and Georgios Gousios. Matching github developer profiles to job advertisements. In *Proceedings of the 12th Working Conference on Mining Software Repositories, MSR '15*, pages 362–366. IEEE Press, 2015.
- [95] René Witte, Yonggang Zhang, and Juergen Rilling. Empowering software maintainers with semantic web technologies. In *The Semantic Web: Research and Applications*, pages 37–52. Springer, 2007.
- [96] Andrea Di Sorbo, Sebastiano Panichella, C Visaggio, Massimiliano Di Penta, Gerardo 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.

- [97] Ebrahim Bagheri and Faezeh Ensan. Semantic tagging and linking of software engineering social content. *Automated Software Engineering*, pages 1–44, 2014.
- [98] Abhishek Sharma, Yuan Tian, and David Lo. Nirmal: Automatic identification of software relevant tweets leveraging language model. In *Software Analysis, Evolution and Reengineering (SANER), 2015 IEEE 22nd International Conference on*, pages 449–458. IEEE, 2015.
- [99] Jinqiu Yang and Lin Tan . Inferring semantically related words from software context. In *Proceedings of the Working Conference on Mining Software Repositories (MSR’12)*, June 2012.
- [100] Jinqiu Yang and Lin Tan. Swordnet: Inferring semantically related words from software context. *Empirical Software Engineering*, 19(6):1856–1886, 2014.
- [101] Matthew J Howard, Samir Gupta, Lori 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*, pages 377–386. IEEE Press, 2013.
- [102] Yuan Tian, Daniel Lo, and Julia 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*, pages 44–53. IEEE, 2014.
- [103] Sen Wu, Zhanpeng Fang, and Jie Tang. Accurate product name recognition from user generated content. In *Data Mining Workshops (ICDMW), 2012 IEEE 12th International Conference on*, pages 874–877. IEEE, 2012.
- [104] Youzheng Wu, Jun Zhao, and Bo 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*, pages 65–72. Association for Computational Linguistics, 2003.

- [105] Khaled Shaalan and Hafsa Raza. Nera: Named entity recognition for arabic. *Journal of the American Society for Information Science and Technology*, 60(8): 1652–1663, 2009.
- [106] Aaron Clauset, Cristopher Moore, and Mark EJ Newman. Hierarchical structure and the prediction of missing links in networks. *Nature*, 453(7191):98–101, 2008.
- [107] David Liben-Nowell and Jon Kleinberg. The link-prediction problem for social networks. *Journal of the American society for information science and technology*, 58(7):1019–1031, 2007.
- [108] Linyuan Lü and Tao Zhou. Link prediction in complex networks: A survey. *Physica A: Statistical Mechanics and its Applications*, 390(6):1150–1170, 2011.
- [109] Eric Gilbert and Karrie Karahalios. Predicting tie strength with social media. In *Proceedings of the SIGCHI conference on human factors in computing systems*, pages 211–220. ACM, 2009.
- [110] Mohammad Al Hasan and Mohammed J Zaki. A survey of link prediction in social networks. In *Social network data analytics*, pages 243–275. Springer, 2011.
- [111] Daniel M Dunlavy, Tamara G Kolda, and Evrim Acar. Temporal link prediction using matrix and tensor factorizations. *ACM Transactions on Knowledge Discovery from Data (TKDD)*, 5(2):10, 2011.
- [112] Rossano Schifanella, Alain Barrat, Ciro Cattuto, Benjamin Markines, and Filippo Menczer. Folks in folksonomies: social link prediction from shared metadata. In *Proceedings of the third ACM international conference on Web search and data mining*, pages 271–280. ACM, 2010.
- [113] David M Blei, Andrew Y Ng, and Michael I Jordan. Latent dirichlet allocation. *the Journal of machine Learning research*, 3:993–1022, 2003.
- [114] A. Sharma, Y. Tian, and D. Lo. Nirmal: Automatic identification of software relevant tweets leveraging language model. In *2015 IEEE 22nd International*

- Conference on Software Analysis, Evolution, and Reengineering (SANER)*, pages 449–458, 2015.
- [115] Stephen Robertson and Hugo Zaragoza. *The probabilistic relevance framework: BM25 and beyond*. Now Publishers Inc, 2009.
- [116] Yoon Kim. Convolutional neural networks for sentence classification. *arXiv preprint arXiv:1408.5882*, 2014.
- [117] Dasha Bogdanova, Cícero Nogueira dos Santos, Luciano Barbosa, and Bianca Zadrozny. Detecting semantically equivalent questions in online user forums. In *Proceedings of the 19th Conference on Computational Natural Language Learning, CoNLL 2015, Beijing, China, July 30-31, 2015*, pages 123–131, 2015.
- [118] Wei Shen, Jianyong Wang, Ping Luo, and Min Wang. Liege: Link entities in web lists with knowledge base. In *Proceedings of the 18th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, KDD '12*, pages 1424–1432. ACM, 2012.
- [119] Rada Mihalcea and Andras Csomai. Wikify!: linking documents to encyclopedic knowledge. In *Proceedings of the sixteenth ACM conference on Conference on information and knowledge management*, pages 233–242. ACM, 2007.
- [120] Xiaohua Liu, Yitong Li, Haocheng Wu, Ming Zhou, Furu Wei, and Yi Lu. Entity linking for tweets. In *ACL (1)*, pages 1304–1311, 2013.
- [121] Julia Preusse, Jérôme Kunegis, Matthias Thimm, Steffen Staab, and Thomas Gottron. Structural dynamics of knowledge networks. In *ICWSM*, 2013.
- [122] Mark EJ Newman. Assortative mixing in networks. *Physical review letters*, 89(20):208701, 2002.
- [123] Sergei Maslov, Kim Sneppen, and Alexei Zaliznyak. Detection of topological patterns in complex networks: correlation profile of the internet. *Physica A: Statistical Mechanics and its Applications*, 333:529–540, 2004.
- [124] Jianhua Lin. Divergence measures based on the shannon entropy. *Information Theory, IEEE Transactions on*, 37(1):145–151, 1991.

- [125] Adam Fourney and Meredith Ringel Morris. Enhancing technical q&a forums with citehistory. In *Proceedings of the Seventh International Conference on Weblogs and Social Media, ICWSM 2013*, 2013.
- [126] Christopher D. Manning, Prabhakar Raghavan, and Hinrich Schütze. *Introduction to Information Retrieval*. Cambridge University Press, New York, NY, USA, 2008. ISBN 0521865719, 9780521865715.
- [127] Vincent D Blondel, Jean-Loup Guillaume, Renaud Lambiotte, and Etienne Lefebvre. Fast unfolding of communities in large networks. *Journal of Statistical Mechanics: Theory and Experiment*, 2008(10):P10008, 2008.
- [128] Mathieu Bastian, Sebastien Heymann, Mathieu Jacomy, et al. Gephi: an open source software for exploring and manipulating networks. *ICWSM*, 8:361–362, 2009.
- [129] Gary Marchionini. Exploratory search: from finding to understanding. *Communications of the ACM*, 49(4):41–46, 2006.
- [130] Claudia Wagner, Philipp Singer, Markus Strohmaier, and Bernardo A Huberman. Semantic stability in social tagging streams. In *Proceedings of the 23rd international conference on World Wide Web*, pages 735–746. International World Wide Web Conferences Steering Committee, 2014.
- [131] Taher H Haveliwala. Topic-sensitive pagerank. In *Proceedings of the 11th international conference on World Wide Web*, pages 517–526. ACM, 2002.
- [132] Antoine Bordes and Evgeniy Gabrilovich. Constructing and mining web-scale knowledge graphs: Kdd 2014 tutorial. In *Proceedings of the 20th ACM SIGKDD international conference on Knowledge discovery and data mining*, pages 1967–1967. ACM, 2014.
- [133] Sébastien Ferré and Alice Hermann. Semantic search: Reconciling expressive querying and exploratory search. In *The Semantic Web–ISWC 2011*, pages 177–192, 2011.

- [134] Erik F Tjong Kim Sang and Fien De Meulder. Introduction to the conll-2003 shared task: Language-independent named entity recognition. In *Proceedings of the seventh conference on Natural language learning at HLT-NAACL 2003-Volume 4*, pages 142–147. Association for Computational Linguistics, 2003.
- [135] Sihem Amer-Yahia, Francesco Bonchi, Carlos Castillo, Esteban Feuerstein, Isabel Mendez-Diaz, and Paula Zabala. Composite retrieval of diverse and complementary bundles. 2014.
- [136] Kevin Gimpel, Nathan Schneider, Brendan O’Connor, Dipanjan Das, Daniel Mills, Jacob Eisenstein, Michael Heilman, Dani Yogatama, Jeffrey Flanigan, and Noah 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*, pages 42–47. Association for Computational Linguistics, 2011.
- [137] Antonio Toral and Rafael Munoz. A proposal to automatically build and maintain gazetteers for named entity recognition by using wikipedia. In *Proceedings of EACL*, pages 56–61, 2006.
- [138] Jun’ichi Kazama and Kentaro 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)*, pages 698–707, 2007.
- [139] Pontus Stenetorp, Sampo Pyysalo, Goran Topić, Tomoko Ohta, Sophia Ananiadou, and Jun’ichi Tsujii. brat: a web-based tool for NLP-assisted text annotation. In *Proceedings of the Demonstrations Session at EACL 2012*, Avignon, France, April 2012. Association for Computational Linguistics.
- [140] Jiang Guo, Wanxiang Che, Haifeng Wang, and Ting Liu. Revisiting embedding features for simple semi-supervised learning. In *EMNLP*, pages 110–120, 2014.
- [141] Mo Yu, Tiejun Zhao, Daxiang Dong, Hao Tian, and Dianhai Yu. Compound embedding features for semi-supervised learning. In *HLT-NAACL*, pages 563–568, 2013.

- [142] Mengqiu Wang and Christopher D Manning. Effect of non-linear deep architecture in sequence labeling. In *IJCNLP*, pages 1285–1291, 2013.
- [143] Radim Řehůřek and Petr Sojka. Software Framework for Topic Modelling with Large Corpora. In *Proceedings of the LREC 2010 Workshop on New Challenges for NLP Frameworks*, pages 45–50, Valletta, Malta, May 2010. ELRA.
- [144] Zellig S Harris. Distributional structure. *Word*, 10(2-3):146–162, 1954.
- [145] Laurens Van der Maaten and Geoffrey Hinton. Visualizing data using t-sne. *Journal of Machine Learning Research*, 9(2579-2605):85, 2008.
- [146] Xavier Glorot, Antoine Bordes, and Yoshua Bengio. Deep sparse rectifier neural networks. In *International Conference on Artificial Intelligence and Statistics*, pages 315–323, 2011.
- [147] Diederik Kingma and Jimmy Ba. Adam: A method for stochastic optimization. *arXiv preprint arXiv:1412.6980*, 2014.
- [148] Tom Kenter and Maarten de Rijke. Short text similarity with word embeddings. In *Proceedings of the 24th ACM International on Conference on Information and Knowledge Management*, pages 1411–1420. ACM, 2015.
- [149] Jean-Philippe Vert, Koji Tsuda, and Bernhard Schölkopf. A primer on kernel methods. *Kernel Methods in Computational Biology*, pages 35–70, 2004.
- [150] Chih-Wei Hsu, Chih-Chung Chang, Chih-Jen Lin, et al. A practical guide to support vector classification. 2003.
- [151] Lili Mou, Ge Li, Lu Zhang, Tao Wang, and Zhi Jin. Convolutional neural networks over tree structures for programming language processing. In *Proceedings of the Thirtieth AAAI Conference on Artificial Intelligence*, 2016.
- [152] Gang Luo, Xiaojiang Huang, Chin-Yew Lin, and Zaiqing Nie. Joint named entity recognition and disambiguation. In *Proc. EMNLP*, 2015.

- [153] Zongcheng Ji, Aixin Sun, Gao Cong, and Jialong Han. Joint recognition and linking of fine-grained locations from tweets. In *Proceedings of the 25th International Conference on World Wide Web*, pages 1271–1281. International World Wide Web Conferences Steering Committee, 2016.
- [154] Nguyen Bach and Sameer Badaskar. A review of relation extraction. *Literature review for Language and Statistics II*, 2007.
- [155] Jun Zhu, Zaiqing Nie, Xiaojiang Liu, Bo Zhang, and Ji-Rong Wen. Statsnowball: a statistical approach to extracting entity relationships. In *Proceedings of the 18th international conference on World wide web*, pages 101–110. ACM, 2009.