Distilling crowd knowledge from software-specific Q&A discussions for assisting developers' knowledge search

Chen, Chunyang

2018

Chen, C. (2017). Distilling crowd knowledge from software-specific Q&A discussions for assisting developers' knowledge search. Doctoral thesis, Nanyang Technological University, Singapore.

<http://hdl.handle.net/10356/75873>

<https://doi.org/10.32657/10356/75873>

NANYANG TECHNOLOGICAL UNIVERSITY

**Distilling Crowd Knowledge from
Software-Specific Q&A Discussions for
Assisting Developers' Knowledge Search**

Chunyang Chen

School of Computer Science and Engineering

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

June, 2018

THESIS ABSTRACT

Distilling Crowd Knowledge from Software-Specific Q&A Discussions for Assisting Developers' Knowledge Search

by

Chunyang Chen

Doctor of Philosophy

School of Computer Science and Engineering

Nanyang Technological University, Singapore

With software penetrating into all kinds of traditional or emerging industries, there is a great demand on software development. Faced with the fact that there is a limited number of developers, one important way to meet such urgent needs is to significantly improve developers' productivity. As the most popular Q&A site, Stack Overflow has accumulated abundant software development knowledge. Effectively leveraging such a big data can help developers reuse the experience there to further improve their working efficiency.

However, the rich yet unstructured large-scale data in Stack Overflow makes it difficult to search due to two reasons. First, there are too many questions and answers within the site, and there may be lingual gap (the same meaning can be written in different languages) between the query and content in Stack Overflow. In addition, the decay of information quality such as misspelling, inconsistency, and abuse of domain-specific abbreviations aggravates the search performance. Second, some higher-order knowledge in Stack Overflow is implicit for searching and it needs certain distillation from existing raw data.

In this thesis, I present methods for supporting developers' information search over Stack Overflow. To overcome the lexical gap and information decay, I also develop an edit recommendation tool to ensure the post quality of Stack Overflow so that posts can be more easily searched by the query. But such explicit information search still requires developers to read, understand and summarize, which is time-consuming. So I propose to shift from the document (information) search to entity (knowledge) search by mining the implicit knowledge from tags in Stack Overflow to render direct answers to developers instead of asking them to read lengthy documents. I first build a basic software-specific knowledge graph including thousands of software-engineering terms and their associations by association rule mining and community detection. Then, I enrich the knowledge graph with more fine-grained relationships i.e., analogy among different third-party libraries. Finally, I combine both semantic and lexical information to infer morphological forms of software terms so that the knowledge graph is more robust for knowledge search.

Acknowledgement

First and foremost, I would like to express my deep gratitude to my advisors Prof. Xing Zhenchang and Prof. Liu Yang, for their guidance, advice, and continuous support throughout my Ph.D. study. They have always been appreciative of new ideas and encouraged critical thinking and technical writing, which have helped me improve my skills. They have been of invaluable support throughout these years in many ways such as atmosphere of freedom, insightful discussions, which has really helped me shape my research and future careers.

I am grateful to Dr Lingfeng, Bao, Dr Yinxing Xue, Sa Gao, Jing Li, Deheng Ye, Guibin Chen, Lei Han, and Mahintham Chandramohan for their valuable advice, suggestions and brainstorming sessions that helped me to improve my research work.

I would like to express appreciation for my friends and colleagues at the Cyber Security Lab (CSL) for their support and encouragement, especially, Dr Bihuan Chen, Dongxia Wang, Zhimin Wu, Xiaofei Xie, Junjie Wang, Xiaoning Du, HongXu Chen, Zhengzi Xu, Ruitao Feng, Yuan Zhou, and Yi Huang. I take this opportunity to thank our laboratory executive Tan Suan Hai for providing technical support at CSL.

I also thank all FYP and Master students working with me, especially, Ke Ming Teong, Yong Ming Leon Lee, Ximing Wang, Kent Ong Long Xiong, Li Yin Tee, Gao Han Toh, Linus Lim Ji Wei, Bernard Wei Jun Chow, Hong Da Koh, Tian Lin from NTU and Qiancheng Wang, Jixuan Cao, Tonghui Yuan, Zhuoqi Qiu, Xi Chen, Yi Huang, Renfei Yang, Yuchen Li, Linzhao Wu, Xu Wang, Jiamou Sun from Australian National University.

I would like to thank my friends and colleagues at Computational Intelligence Lab (CIL), especially Zhu Sun, Jianjun Zhao, Mengchen Zhao, and Chit Lin Su for their support. I am also thankful to Kesavan Asaithambi for his technical assistance in CIL.

Finally, I want to express my utmost gratitude to my family, my parents, my wife, and my baby boy. Their care and company are precious treasures in my life, and their support and trust help me finish my PhD study.

Contents

1	Introduction	1
1.1	Motivations and Goals	2
1.2	Main Works and Contributions	3
1.2.1	Software-specific text retrieval	4
1.2.2	Software-specific knowledge graph for entity-centric search	4
1.3	Thesis Outline	5
2	Literature Review	7
2.1	Information Retrieval in Software Engineering	7
2.1.1	Quality assurance of Stack Overflow for retrieval	8
2.2	Software-specific Knowledge Graph for Search	10
2.2.1	Constructing software-specific knowledge graph	10
2.2.2	Mining analogical libraries to enrich the knowledge graph	11
2.2.3	Mining morphological forms to enhance knowledge graph	13
2.3	Deep Learning in Software Engineering	14
2.3.1	Deep learning for source code	15
2.3.2	Deep learning for software documents	16
3	A Deep Learning Approach to Assist Collaborative Editing in Q&A Sites to Enhance the Text Quality	18
3.1	Introduction	18
3.2	Formative Study of Post Editing in Stack Overflow	22
3.2.1	What has been edited?	23
3.2.2	Who edited posts?	24
3.2.3	What are post edits about?	25
3.2.4	What is the scale of changes that post edits involve?	26
3.3	Assisting Sentence Editing	27
3.3.1	Approach overview	27
3.3.2	Collecting the corpus of original-edited sentence pairs	28
3.3.3	Character-level RNN Encoder-Decoder Model	30
3.3.4	Normalizing domain-specific rare words	33
3.3.5	Implementation	34
3.4	The Quality of Recommended Sentence Edits	34
3.4.1	Dataset	34
3.4.2	Baseline methods	35

3.4.3	Evaluation metric	35
3.4.4	Evaluation results	37
3.4.4.1	What is the quality of recommended sentence edits by our method? How much improvement can it achieve over the baseline methods?	38
3.4.4.2	In which cases does our model recommend low-quality sentence edits?	39
3.5	Assisting Novice Post Editors in Post Editing	41
3.6	Feedbacks of Trusted Post Editors	42
3.7	Discussion	44
3.7.1	Unintended consequences of the edit assistance tool	44
3.7.2	Limitations	45
3.8	Conclusion	46
4	Mining Knowledge Graph from Stack Overflow to Understand Technology Landscape	47
4.1	Introduction	47
4.2	The Approach	50
4.2.1	Mining technology associations	51
4.2.2	Detecting technology communities	52
4.2.3	Determining technology categories	52
4.2.4	Summarizing technology activity	53
4.2.5	Visualizing TAN	53
4.3	The TechLand System	54
4.3.1	Augmenting search with information scents	54
4.3.2	Inspecting aggregation of correlated technologies	56
4.3.3	Comparing technology adoption indicators	57
4.3.4	Suggesting relevant web resources	58
4.3.5	The usage of TechLand website	59
4.4	Empirical Study	60
4.4.1	Dataset	61
4.4.2	RQ1: Size and modularity of technology communities	61
4.4.3	RQ2: Coverage of tags and questions	62
4.4.3.1	Coverage of questions	63
4.4.3.2	Coverage of tags	63
4.4.4	RQ3: Semantic distance of technology associations	65
4.5	User Study	66
4.5.1	Tasks	66
4.5.2	Participants	67
4.5.3	Experiment procedures	68
4.5.4	Results	68
4.5.4.1	System performance	69
4.5.4.2	System usability	70

4.5.4.3	User feedback	70
4.5.5	Limitations	71
4.6	Conclusion	71
5	Mining Analogical Libraries from Q&A Discussions	72
5.1	Introduction	72
5.2	The Approach	77
5.2.1	Mining categorical information	78
5.2.2	Mining relational information	80
5.2.3	Learning tag embeddings	82
5.2.4	Building analogical-libraries knowledge base	83
5.2.4.1	Obtain analogical-library candidates	84
5.2.4.2	Refine initial results	85
5.2.5	Assisting analogical-libraries comparison	86
5.2.5.1	Extract comparison questions	86
5.2.5.2	Extract comparison answer snippets	87
5.3	Tool Support	89
5.3.1	Tool description	89
5.3.2	Search engine optimization	91
5.4	Accuracy Evaluation	93
5.4.1	The accuracy of tag categorization	93
5.4.2	The semantic distance of tag correlations	94
5.4.3	The accuracy of analogical-libraries recommendation	95
5.4.3.1	Evaluation procedure and metrics	95
5.4.3.2	Accuracy results	97
5.4.3.3	Analysis of inaccurate recommendations	98
5.4.4	The relevance of comparison questions & answer snippets	99
5.5	Usefulness Evaluation	100
5.5.1	Experimental setup	101
5.5.2	Results	101
5.6	Field Study	103
5.6.1	Site traffic statistics	103
5.6.2	Most interested libraries, languages and comparisons (RQ1)	105
5.6.3	Analogical libraries within or across languages (RQ2)	105
5.6.4	Usefulness of tagWiki and asking trend (RQ3)	107
5.6.5	Real developers' visits (RQ4)	109
5.7	Conclusion	110
6	Unsupervised Software-Specific Morphological Forms Inference from Informal Discussions	111
6.1	Introduction	111
6.2	Approach	114
6.2.1	Dataset	115
6.2.2	Preprocessing input corpuses	116

6.2.2.1	Text cleaning	116
6.2.2.2	Phrase detection	116
6.2.3	Building software-specific vocabulary	117
6.2.4	Learning term semantics	118
6.2.5	Extracting semantically related terms	119
6.2.6	Discriminating synonyms & abbreviations	121
6.2.6.1	Discriminating morphological synonyms	121
6.2.6.2	Discriminating abbreviations	123
6.2.7	Grouping morphological synonyms	124
6.3	Evaluation of Our Thesaurus	125
6.3.1	The coverage of software-specific vocabulary	125
6.3.2	Abbreviation coverage	127
6.3.3	Synonym coverage	128
6.3.4	Human evaluation	129
6.4	Usefulness Evaluation	130
6.4.1	Background	131
6.4.2	Experiment setup	131
6.4.2.1	Dataset	131
6.4.2.2	Compared methods	131
6.4.2.3	Ground truth and evaluation metrics	132
6.4.3	Results	133
6.5	Applications from Our Approach	134
6.5.1	Software-specific spelling correction	134
6.5.1.1	Data collection	134
6.5.1.2	Experiment setup	135
6.5.1.3	Experiment results	136
6.5.2	Software-specific text wikification	136
6.5.3	Identify tag synonyms on Stack Overflow	139
6.5.4	Building software dictionary	140
6.6	Conclusions	142
7	Conclusions and Future Research	143
7.1	Summary of Completed Work	143
7.2	Future Work	145
	Bibliography	148
	Appendices	171

List of Figures

1.1	The overview of the motivation and goals in this thesis	3
1.2	The roadmap of this dissertation	5
3.1	An example of post edit in Stack Overflow	19
3.2	The numbers of posts and three kinds of post edits in each year in Stack Overflow	23
3.3	The word cloud of the top 50 most frequent words appearing in edit comments (the words are aligned alphabetically from left to right) . . .	24
3.4	The count of original-edited post title and body in different similarity range	26
3.5	The overall workflow of the edit assistant tool	27
3.6	Basic RNN model	30
3.7	The RNN encoder-decoder model	31
3.8	The process to deal with domain-specific rare words	33
3.9	The performance of different methods dealing with sentences of different length (the number of characters of the sentence)	40
3.10	Part of the results of post edits by our model	42
3.11	The survey results	44
4.1	Post #1710476 in Stack Overflow	48
4.2	The general TAN mined at minimal support 0.0007 and minimal confidence 0.15	50
4.3	POS tagging and phrase chunking results of the definition sentence of the tag <i>Matplotlib</i>	53
4.4	Augmenting technology landscape inquiries with four pieces of information that are related to the search terms: 1) the definition of the technology given by Stack Overflow, 2) the trend of community interest in the searched technology, 3) a set of related libraries, languages and concepts, 4) an interactive knowledge graph of technology correlations.	55
4.5	The TechLand web interface. 1) The technology definition and more details can be obtained on TagWiki. 2) The monthly asking trend in Stack Overflow. 3) The overall technology associative network and click year to view the TechLand graph for a year. 4) The top-voted questions tagged with <i>sockets</i> . 5) A Java code snippet for <i>sockets</i> 6) The most-referenced links whose anchor text matches tag name or synonyms. . .	56
4.6	The asking trend comparison among <i>d3.js</i> , <i>matplotlib</i> , <i>ggplot2</i> and <i>mschart</i>	58
4.7	The Google Analytics of our website	60

4.8	The coverage of the general TAN	63
4.9	The impact of minimal confidence on the size and modularity of the general TAN	64
4.10	Average score of SUS results	70
5.1	An example of analogical-library question in Stack Overflow.	73
5.2	The similar context of Python's nltk and Java's opennlp in tag sentences	74
5.3	The overview of our approach	77
5.4	POS tagging and phrase chunking results of the definition sentence of the tag <i>iOS</i>	78
5.5	The continuous skip-gram model. It predicts the surrounding words given the center word.	82
5.6	The screenshot of our website <i>SimilarTech</i> and the important elements in the website	89
5.7	Two example search in Google and our page ranks the first in the search results	92
5.8	Recommendation accuracy using the combination of different kinds of knowledge	97
5.9	An example analogical question and its answers in Stack Overflow. Relevant libraries are often hyperlinked in the answers.	100
5.10	The traffic of our website from Google Analytics	104
5.11	(a) The top-10 most-frequently visited libraries in our website. (b) The number of libraries in each language that have been visited and the total number of visits. (c)The top-10 most-frequently visited library comparisons	105
6.1	The morphological forms of “regular expression” (blue) and “javascript” (red) in three Stack Overflow posts	111
6.2	The overview of our approach	115
6.3	The continuous skip-gram model. It predicts the surrounding words given the center word.	118
6.4	An illustration of term representations in the word embedding space, two-dimensional projection using PCA.	120
6.5	The synonym graph: the edge represents the synonym relationship and a connected component represents a group of synonyms (zoom-in to view the details)	124
6.6	The tags and their different forms in the question	132
6.7	The average percentage of tags appearing in question text by different normalization methods	133
6.8	An URL and its anchor text in post #5956115	137
6.9	The Precision@k of the standard URL description by different normalization methods	138
6.10	The screen shot of our website	141
6.11	The screen shot of the Firefox extension in use	141
7.1	The overview of finished works and future works	145

List of Tables

3.1	Examples of small sentence edits in Stack Overflow	19
3.2	GLEU score for different edits	36
3.3	The performance of different methods for sentence editing	37
3.4	Examples of sentence edits by different methods	37
4.1	Task Descriptions (PL denotes programming language)	67
4.2	The mean (standard deviation) of the three measures, * denotes $p < 0.1$ and ** denotes $p < 0.05$	69
5.1	Example questions about the comparison of the two analogical libraries in Stack Overflow	86
5.2	Example sentences in Stack Overflow answers about the comparison of the two analogical libraries	87
5.3	Example analogical questions and their answers from Stack Overflow and analogical libraries recommended by our method	102
5.4	Language migration statistics	107
5.5	The number of visits from big companies	109
5.6	The comparison of visits between developers from big companies and the rest	109
6.1	Morphological forms of <i>visual c++</i>	112
6.2	Example Abbreviations and Synonyms	123
6.3	The coverage of terms in SOTag and CPtag datasets	125
6.4	The coverage of synonyms in three methods.	129
6.5	Misspelling examples in our thesaurus	135
6.6	Performance comparison for spelling correction	136
6.7	The anchor text and its frequency for the URL	137
6.8	Exmaples of semantically related techniques	140

Chapter 1

Introduction

With software penetrating all kinds of traditional or emerging industries, and rapid development of information technology, there is a great demand on software development. In addition, as the task gets more and more complicated, the corresponding software also becomes more and more complex¹. For example, some software are implemented with millions of lines of source code such as Windows², Jet control system³. However, due to the limited number of developers, there is an urgent need to boost developers' productivity.

During the evolution of software engineering, a lot of software development experience has been recorded in different software repositories such as code in GitHub, Q&A discussions in Stack Overflow, bug reports in Bugzilla, software vulnerabilities in CVE database, etc. These heterogeneous repositories created by millions of developers serve as the big data of software engineering practices. Reusing the experience in the big data can help avoid re-implementing the wheels, which improves developers' productivity.

Stack Overflow is currently the most popular Q&A (Question and Answer) site about programming. It is where developers can ask technical questions, and the answer to those questions may benefit other developers with similar questions. Dated to May

¹<https://informationisbeautiful.net/visualizations/million-lines-of-code/>

²<https://www.quora.com/How-many-lines-of-code-does-Windows-10-contain>

³https://insights.sei.cmu.edu/sei_blog/2015/09/managing-software-complexity-in-models.html

2018, there are more than 16 million questions, 24 million answers and 8.8 million users within the site. A great amount of software development knowledge are embedded in these technical discussions, ranging from detailed API usage pattern, to general IDE comparisons. Every month, 50 million people visit Stack Overflow to learn and share. It is estimated that 21 millions of them are professional developers and university-level students⁴. Stack Overflow has so far emerged as an invaluable resource for developers around the world. My research focuses on mining Stack Overflow to assist developers for knowledge search.

1.1 Motivations and Goals

Although Stack Overflow contains a lot of programming knowledge, it is not easy for developers to obtain such knowledge due to two barriers of the searching (as seen in Figure 1.1).

- **T1. Challenges to explicit search** In Stack Overflow, each post may contain not only natural-language descriptions, but also code snippets. Such unstructured and domain-specific data brings many challenges to efficient search. First, there may be lexical gap between the query and posts because different developers may use entirely different words to express the same meaning. Second, the decay of information quality such as misspelling, grammar errors, inconsistency, and abuse of abbreviations not only significantly increases the difficulty to read or understand posts, but may also cause the mismatch of keywords, leading to bad search results.
- **T2. Challenges to search implicit knowledge** Apart from explicit questions and answers, some knowledge in Stack Overflow is implicit for searching. For example, tens of thousands of software tools and libraries are available for software development. Faced with a wide range of software technologies, they often wonder – what are the best technologies for a particular task? What technologies are related to certain ones? For example, although *JFreeChart* and *ggplot2*

⁴<https://insights.stackoverflow.com/survey/2018/>

FIGURE 1.1: The overview of the motivation and goals in this thesis

are both plotting libraries, they are never explicitly mentioned as counterparts in Stack Overflow. Then for developers who are looking for similar libraries to *JFreeChart* in *R* language, they may miss the results. Developers conventionally resort to search engines which return 10 blue links containing some tutorials, blogs or Q&A posts. However, these expert articles are very likely to be out-of-date, opinion-based and isolated from other related or state-of-the-art technologies.

1.2 Main Works and Contributions

In this thesis, I am trying to support efficient information and knowledge search in Stack Overflow, and five main works are conducted to solve the aforementioned problems and technical challenges.

1.2.1 Software-specific text retrieval

To make software text easier to understand and index by search engine, I develop an edit-assistance tool for identifying quality issues in Stack Overflow posts and recommending sentence editing for correction. I formulate the sentence editing task as a machine translation problem, in which an original poor-quality sentence is “translated” into an edited better-quality sentence. Based on the millions of post edits from Stack Overflow, I implement a character-level Recurrent Neural Network (RNN) encoder-decoder model to correct mistakes in software-specific text.

1.2.2 Software-specific knowledge graph for entity-centric search

But such explicit information search still requires developers to read, understand and summarize, which is time-consuming. Thus, I propose to shift from the original document search to entity search i.e., from information search to knowledge search. Inspired by the fact that Stack Overflow users tag their questions with the main technologies that the questions revolve around, I first build a basic knowledge graph by applying association rule mining and community detection methods to mine their relationships from millions of Stack Overflow question tag. This is the first proposed software-specific knowledge graph, and it can render direct answers to developers’ search such as what technologies to finish certain task, what concepts related to other technologies, etc.

Then I further dive into enriching it with more semantic relationships. A semantic relation in which I am particularly interested in is the analogical relation between third-party libraries. Third-party libraries are an integral part of many software projects. Thus, when developers are migrating their software to other platforms or languages to meet the market need, they have to spot the analogical libraries in the target platform or language. I adopt the word embedding method to embed all Stack Overflow tags into a vector which captures the technology semantics. Then I get the relational knowledge between library and language by association rule mining, and categorical knowledge

FIGURE 1.2: The roadmap of this dissertation

from tag description by natural language processing. By combining relational, categorical information of technologies and the technology semantic, I successfully build a knowledge base of analogical libraries to enrich the knowledge graph.

I then notice a “notorious” phenomenon in my work on mining Q&A discussions, i.e., the wide presence of abbreviations, synonyms and misspells, which may negatively affect the robustness of entity-centric search based on the knowledge graph. To solve that problem, I combine both semantic (from the word embedding) and lexical information (software-specific naming conventions) to infer morphological forms of software terms and then enrich the knowledge graph with a large number of commonly used abbreviations and synonyms.

1.3 Thesis Outline

Figure 1.2 shows the roadmap for this dissertation. The roadmap describes the sequence of my research work and thereby reveals the train of my thoughts on software understanding. The remaining of the thesis is organized as follows:

Chapter 2 presents the literature review of this thesis including background, related works and state-of-the-art techniques.

In Chapter 3, I design and develop a RNN encoder-decoder model [1] to automatically correct the minor issues of the posts in Stack Overflow. It not only assures the quality of the site, but also make the content in Stack Overflow more easily to be searched.

In Chapter 4, I construct a basic software-specific knowledge graph [2, 3] for entity search.

In Chapter 5, I enrich the knowledge graph with a more fine-grained relationship i.e., analogical libraries [4, 5] across different languages to support software migration. And it can be further investigated for selection of third-party libraries [6].

Chapter 6 describes a method [7] to extract synonyms, abbreviations for entities in the knowledge graph. Such a thesaurus make our entity search based on the knowledge graph more robust in face of naming variations.

Chapter 7 concludes the works in the thesis, and proposes the potential directions in the future.

Chapter 2

Literature Review

Stack Overflow has been the focus of many studies¹, including discovering topics and trends of developers' discussions [8, 9], predicting answer quality [10] or user participation [11], identifying experts [12], analyzing social interactions inside the cooperative community of Stack Overflow [13, 14], and studying technology trends [8, 10]. These studies show that Stack Overflow is a high-quality knowledge repository of developers' thoughts, needs and practices. As my thesis mainly focus on the both explicit information retrieval and implicit knowledge retrieval based on knowledge graph, I introduce the related work about information retrieval in software engineering in Chapter 2.1 and constructing software-specific knowledge for retrieval in Chapter 2.2. In addition, as deep learning is the backbone method of most works in this dissertation, I also describe the related works about using deep learning in software engineering domain in Chapter 2.3.

2.1 Information Retrieval in Software Engineering

Information retrieval is an important task in Software Engineering and it has been mainly studied from two aspects, code search and documentation search. Many code

¹A community-curated list of publications using Stack Overflow data can be found at <http://meta.stackexchange.com/q/134495>.

search engines have been proposed based on program patterns [15], test cases [16], program semantics [17], and user feedbacks [18]. For software documentations, Siddharth et al [19] build a plugin to find API documentations for APIs mentioned in Q&A discussions. Robillard and Chhetri [20] can recommend fragments of API documentation potentially important to a programmer using an API. Information retrieval has also been used in bug localization [21], traceability recovery [22], and feature location [23].

With the advent of Web 2.0, Q&A websites such as Stack Overflow have become an important information source for developers. Hence, much work has been carried out on Q&A data, such as ranking answers based on user feedback [24] and automatically answering interrogative questions [25] (e.g., how to and why questions). Compared with these works, we focus on bilingual information retrieval in Q&A site which is more challenging than monolingual analysis.

Some work about searching has also been done on cross-lingual issues in Software Engineering, such as bug localization between Chinese and English [26], traceability recovery between Italian and English [27]. Xu et al. [28] propose a domain-specific bilingual question retrieval system which customizes the general translation by finding software-specific terms in a domain specific corpus to enhance the translation performance. These existing methods is rule-based, relying on human-engineered features and general machine translation system.

2.1.1 Quality assurance of Stack Overflow for retrieval

The decay of information quality such as misspelling, grammar errors, inconsistency, and abuse of abbreviations make it difficult for users to search related content in the web. So much research has been carried out to correct spelling and grammar errors using machine learning techniques. Junczys-Dowmunt and Grundkiewicz [29] adopt phrase-based Statistical Machine Translation (SMT) method for automatic grammar error correction. Mizumoto and Matsumoto [30] and Yuan et al. [31] propose a ranking method to rank the SMT's recommendations of grammar error corrections. However,

SMT focuses on source-target phrase pairs without effective modeling sentence context. Furthermore, SMT consists of components that are trained separately and then combined [32]. Compared with traditional SMT methods, Neural Machine Translation (NMT), such as RNN-based methods [33, 34], models sentence context and all the NMT components are trained jointly to maximize the performance. Especially, NMT is appealing for Grammatical Error Correction (GEC) tasks as it may correct erroneous sentences that have not been seen in the training data. Therefore, our edit assistance tool adopts a RNN encoder-decoder model [35].

Another big limitation of existing NMT methods is the lack of editing data for training deep learning methods. The Helping Our Own task [36] contains 1264 edits for model training and 1057 edits for testing. The CoNLL-2014 task [37] on GEC contains 57151 edited sentence pairs for training and 1312 for testing which are collected from essays written by students at National University of Singapore. To mitigate the lack of data, Liu et al. [38, 39] propose different ways of artificial error generation, but such generated edits may differ from the real data. We are the first to leverage the big data of post edits in Q&A sites to train an edit assistance tool for sentence correction. And our training data is 100 times larger than the largest dataset of existing work [40].

Other researchers also adopt deep learning methods for grammatical error detection [38, 41] and sentence correction [40, 42, 43]. Although these methods obtain better performance than traditional SMT methods, they cannot effectively deal with the three data characteristics of Stack Overflow post edits. First, existing methods are designed for general English text. They cannot handle domain-specific rare words, such as URLs, API calls and variable names in Stack Overflow posts. Second, existing methods consider only general language errors that are only part of post edits in Stack Overflow. They cannot handle the format change such as HTML markdown and domain-specific naming convention. Third, existing methods deal with word- or phrase-level correction, but the majority of post edits involves only minor changes of post content at character level.

2.2 Software-specific Knowledge Graph for Search

Web users are expecting direct answers to their web search, rather than links to documents [44]. Direct answers to web search require organizing the web information into the knowledge graph of linked entities. Mining knowledge graphs from structured (e.g., Freebase, Linking Open Data, DBpedia) and unstructured (e.g., wikipedia, social media) documents is a vibrant area in database and mining research [45–47]. Knowledge-graph based applications, such as query understanding and reformulation [48], entity profiling [49], exploratory search [50], serendipitous search [51] have also been actively researched.

2.2.1 Constructing software-specific knowledge graph

Tagging supports the categorization of information using user-defined, open-ended vocabularies, as opposed to predefined, fixed taxonomies. It is used by many social computing systems, in which users tag objects such as web sites (e.g., Delicious), photos (e.g., Flickr), research papers (e.g., Connotea), software projects (e.g., Freecode, Maven), and questions (e.g., Stack Overflow, Quora). Furthermore, tagging has also been integrated in software development process. Storey et al. [52] develop the TagSEA tool that uses the ideas of social tagging to support collaboration in asynchronous software development. Treude and Storey [53] show that tagging of work items in IBM Jazz can improve team-based software development practices. Nasehi et al. shows that the main technologies that the question revolves around can usually be identified from question tags [54]. These works shows that tags used in software repositories represent important knowledge units which are potential to be adopted as entities in knowledge graph.

Many studies have shown that structured knowledge can emerge from social tagging systems [55–57]. Hierarchical clustering techniques have been applied to induce taxonomies from collaborative tagging systems [58, 59], and from software project hosting

site Freecode [60]. Schmitz analyzes association rule mining results to infer a subsumption based model from Flickr tags [61]. Sanderson and Croft [62] analyze the co-occurrence of words to derive concept hierarchies from text. Tian et al. [63] construct a software-specific related words by computing the word co-occurrence weights in a corpus of Stack Overflow questions. Yang and Tan [64] infer semantically related words from software source code. The goal of these two works is to build software-specific dictionary to determine associative meanings between software-specific technical terms. Different from these taxonomy mining techniques, our technology associative network is a complex, non-hierarchical, and multi-faceted network.

Although software engineering community has a long history of studying graphical software models [65, 66], the concept of knowledge graph, the relevant mining techniques, the application of knowledge graph in software engineering context have not been widely adopted in this community. Our work attempts to mine knowledge graph from Stack Overflow question tags.

2.2.2 Mining analogical libraries to enrich the knowledge graph

Within the knowledge graph, we find that the third-party library is a kind of important knowledge unit. Language migration is a common phenomenon for developers as they may have to switch from one programming language to another according to the task requirements. The biggest challenge is usually the code and library migration, rather than learning a new language itself². Many researchers have proposed methods to overcome the code migration challenge, such as code mapping [67], function mapping [68], and API migration [69, 70]. In contrast to these code-level migration approaches, our approach supports library-level migration.

Thung et al. [71] analyze the library co-occurrence patterns in software projects to recommend relevant libraries for a software project. Teyton et al. [72, 73] analyze the evolution of projects' dependencies on third-party libraries to recommend libraries that can replace an existing library in a software project. Different from these approaches, our

²<http://stackoverflow.com/questions/212151/>

approach does not rely on the information about the projects' dependencies on third-party libraries. Instead, we mine analogical libraries from the crowdsourced knowledge in domain-specific Q&A sites (such as Stack Overflow). Furthermore, existing approaches are limited to recommend libraries for the same programming language, while our system can recommend alternative, comparable libraries across different programming languages.

Our approach is motivated by the recent success of using word embeddings techniques to solve semantic and syntactic analogy tasks in NLP applications [74, 75]. We propose to learn tag embeddings from a corpus of tag sentences derived from Stack Overflow questions, and solve the problem of finding analogical libraries using the vector arithmetic of the resulting tag embeddings. Different from English text in common NLP problems, our tag sentences are short and lack of linguistic rules and notions. Inspired by the recent work by Xu et al. [76] and Zhou et al. [77], we propose to incorporate relational and categorical knowledge of tags into tag embeddings to improve the accuracy of analogical-libraries reasoning tasks. Different from [76, 77] in which relational and categorical knowledge are provided by human experts, our approach mines domain-specific knowledge automatically from Q&A discussions and community wikis on Stack Overflow. The extracted analogy relationships are then used to enrich our existing knowledge graph.

Word embedding has recently been adopted in analyzing software engineering data. Vu et al. [78, 79] mine user opinions in the mobile application reviews by extracting semantically related words by word embedding. Ye et al. [80] improve the information retrieval in software engineering text by using word embeddings to measure document similarities. Nguyen et al. [81] characterizes API elements with word vector representation and then map word vectors for code migration [82]. Word embedding can also be incorporated into deep learning method like Convolutional Neural Network for different tasks in software engineering such as predicting semantically linkable knowledge [83] and cross-lingual information retrieval [84]. Different from these works, we not only encode tags with word embedding techniques, but also exploit relational similarity of tag embeddings to reason about analogical libraries.

It is also worth mentioning some related non-academic projects. SimilarWeb³ is a website that provides both users engagement statistics and similar competitors for websites and mobile applications. AlternativeTo⁴ is a social software recommendation website in which users can find alternatives to a given software based on user recommendations. These websites can help regular web users to find similar or alternative websites or software applications. But their content is not useful for domain-specific information needs of software developers, for example, to find analogical libraries for different programming languages. In contrast, our web application is built on software-engineering data and is specifically designed for software developers.

2.2.3 Mining morphological forms to enhance knowledge graph

When developing software, developers often use abbreviations and acronyms in identifiers and domain-specific terms that must be typed in source code and documentation. Informal discussions on social platforms (e.g., Stack Overflow) also contains many abbreviations, synonyms and misspellings such as *obj-c* for *objective-c*, and *d3js* for *d3.js*. This phenomena challenges the effectiveness of knowledge graph based techniques for searching and rendering direct answers.

To reliably infer morphological forms of software-specific terms in a broad range of informal discussions, we must analyze semantic relatedness of software-specific terms and morphological forms. One way is to develop a software-specific thesaurus like the WordNet [85] for general text, and the BioThesaurus [86] for biology domain. In fact, most of identifier expansion approaches [87–91] use in a way or the other external dictionary, such as general English dictionary, domain-specific ontology and/or a list of well-known abbreviations to rank the relatedness of abbreviation and expansions. Although such dictionary is useful for reasoning word relatedness, it requires significant time and efforts to build and it is difficult to scale it up and keep it up-to-date. This is why we do not want our approach to rely on such dictionary. Indeed, our work presents

³www.similarweb.com/

⁴<http://alternativeto.net/>

an automatic approach to mine a high-quality thesaurus of software-specific terms and morphological forms by analyzing the text data alone.

Several attempts have been made to automatically infer semantically related terms in software engineering text [63, 90, 92–95]. However, the proposed techniques make some important assumptions about the text, for example, relying on project-specific naming or documentation convention [94, 96], or lexical difference patterns between sentences [95], or the availability of certain contextual information (e.g., dictionary, question tags) [60, 90, 97]. Such assumptions lack the generality for other data. Unlike existing approaches, we resort to unsupervised word representations (i.e., word embeddings) [74] to capture word semantics from a large corpus of unlabeled software engineering text. Recently, word embeddings have been successfully applied to various software engineering problems involving text data, such as document retrieval [80, 98], software-specific entity recognition [99], analogical library recommendation [5, 100], prediction of semantically related posts [101]. Our goal is different: based on semantically similar software-specific terms, we infer a group of morphological forms which represent a distinct concept in informal discussions.

2.3 Deep Learning in Software Engineering

During the last few years, deep learning has displayed its power in different areas such as image classification [102], text understanding [103], speech recognition [104], etc. With the accumulation of big software data such as software code, software documents, many researchers also try to apply deep learning [80, 105, 106] into software-engineering domain to assist developers with different tasks. As deep learning is one of the most important technologies adopted in this dissertation, we introduce related works and technology background about using deep learning in software code and documents.

2.3.1 Deep learning for source code

Recently, some researchers have explored the possibility of applying deep learning techniques to source code. Automatically generating the code is the dream to alleviate developers' burden for software developments, hence many researchers try to leverage deep learning for code generation. Based on the test cases, Balog et al [107] use the neural network's predictions to augment search techniques from the programming languages community to solve programming competition-style problems. Alexandru and Carol [108] guide code synthesis with deep neural network. Some other works also apply deep learning to generate code based on different inputs. With the popularity of deep learning, many research papers are published about it, but without releasing the detailed code. To help other researchers easily replicate deep learning papers, Sethi et al [109] propose a neural network to generate the code automatically from the research papers. Inspired by the recent progress of image captioning, Chen et al [110] and Beltramelli [111] generate UI skeleton code, given the mobile UI design images.

Apart from code generation, neural networks are also applied to code completion. Raychev et al [112] applied the RNN language to complete the partial programs with holes, and White et al [113] train the RNN model in large-scale projects in Github and show its effectiveness in predicting source code tokens. Another typical application is to obtain code representation via deep learning. For example, Peng et al [114] propose to learn code vectors with deep learning. Mou et al [115] adopt convolutional neural network (CNN) over AST trees for code classification.

API is an encapsulate block of source code, and it can be reused in different projects. However, due to the large number of existing API and the lexical gap between the query and API description, it is difficult for developers to quickly search what they want. So to infer similar API, Nguyen et al [105] adopt word embedding to obtain the vector representation of API by training the model in a large-scale code corpus collected from GitHub. Gu et al [106] collect the comments within the method declaration and corresponding API call sequence as the training data. They adopt the sequence-to-sequence model to generate Java API sequence given the natural language descriptions.

Therefore, developers especially novices who are not familiar with existing APIs can benefit a lot from that tool.

2.3.2 Deep learning for software documents

Although source code is the core in software engineering, software documents are also important because it plays a role as glue among different developers. Compared with millions lines of source code, it is much easier to understand the code according to its corresponding documents. Therefore, much research has been carried out to assist developers with searching software documents. Lam et al [116] combine the neural network with information retrieval to localize code files for bug reports. Ye et al [80] aggregate word embedding to encode software documents into high-level representation to improve document search in software engineering domain. Guo et al [117] adopt LSTM based sequence-to-sequence model to search the code file by inputting the natural-language requirement text. Deep learning is also used for similarity measurement among software documents. For example, Yang et al [118] combine word embedding with information retrieval to obtain similar bug reports. Xu et al [83] adopt CNN to learn word-level and document-level features to predict semantically linkable knowledge units on Stack Overflow, and they further adopt the text summarization based on deep learning [119] to summarize answers to technical questions in software-specific Q&A site.

Neural networks are also utilized to help mine software vulnerability to enhance the software security. Han et al [120] learn to predict the severity of software vulnerability using CNN on vulnerability descriptions. They [121] then build a knowledge graph of CVEs and reason potential undetected links among existing CVEs by using the knowledge graph embedding. Yuan et al [122] propose the basic fully-connected neural network based on hundreds of extracted features for malware detection, while Kolosnjaji et al [123] adopt more advanced method, CNN to model the malware system call sequence to infer the malwares.

All of these works explore different aspects in software engineering to assist developers to work more productively with deep-learning tools. This dissertation is highly inspired by related works mentioned above, and most works in this dissertation also adopt the deep learning approach such as word embedding, RNN, RNN encoder-decoder for different tasks like ensuring software-text quality, obtain tag embedding, and mining synonyms and abbreviations.

Chapter 3

A Deep Learning Approach to Assist Collaborative Editing in Q&A Sites to Enhance the Text Quality

3.1 Introduction

Due to the casualness of software developers, the software text is always filled with misspellings, grammar errors, abuse of abbreviations, and the lack of consistent format. Such kind of quality decay not only leads to the difficulty of others' understanding of software text such as code documents, comments, technical discussions which hinder the software development. It also negatively influences the content searching in Stack Overflow, especially when the key software-specific terms are misspelled like *javascript* to *iavascrip*.

In this chapter, we focus on the software text quality of Stack Overflow which is the most popular Question and Answering (Q&A) site for software programming. It hosts a community of millions of developers who share and learn software programming knowledge. An important reason for the popularity of Stack Overflow is that the content of Stack Overflow posts has been well maintained by the community. After a question or answer (referred to as a post in Stack Overflow) is posted, it can be self-edited by the

XPATH XPath - Selection TD next to the selected xpathXPath table which contains SPAN

I have a table ~~in~~and I'm trying to get data from via xpath. A simple example of the table looks like this:

horse	id1	id2	id3	id4
abc	1	1	1	1
123	2	2	2	2
cba	3	3	3 3	
321	4	4	4	4

What I want to do is look at column ~~id3~~ id3 and find the row that contains the span code (in this case ~~it's~~ row 3). Once I have this I would like to get the value in column 1 of that row (the one that span is on) which would be cba.

Can ~~Anyone~~anyone help?

FIGURE 3.1: An example of post edit in Stack Overflow

Original Sentence	Edited Sentence	Editing Reason
1 I need to get the last char of a <u>srtng</u> .	I need to get the last char of a <u>string</u> .	Spelling
2 Is it possible to this?	Is it possible to <u>do</u> this?	Grammar
3 Can you suggest me	Can you suggest me?	Punctuation
4 Any ideas how to fix it?	Any ideas how to fix it?	Space
5 <u>how</u> can I accomplish this?	How can I accomplish this?	Capitalization
6 My problem is the when I click on the <u>OK</u> button, nothing happens.	My problem is the when I click on the <u>`OK`</u> button, nothing happens.	Annotation
7 <u>EDIT</u> : Sorry, I should have inserted the term "cross browser" somewhere.	<u>**EDIT**</u> : Sorry, I should have inserted the term "cross browser" somewhere.	Annotation
8 1) How to connect to SVN server from java?	<u></u> How to connect to SVN server from java? <u></u>	HTML
9 I want an <u>Apple</u> script that refreshes a certain song in iTunes from file.	I want an <u>AppleScript</u> that refreshes a certain song in iTunes from <u>a</u> file.	Spelling
10 I am trying to parse a set of <u>xml</u> files.	I am trying to parse a set of <u>XML</u> files.	Capitalization
11 Use javascript function isNaN.	Use JavaScript function isNaN.	Capitalization

TABLE 3.1: Examples of small sentence edits in Stack Overflow

post owner (i.e., the question asker or answerer) and/or collaboratively edited by some post editors¹ (i.e., users other than the post owner). As of December 11, 2016, 12.3 million posts (i.e., about 37% of all 33.5 million posts) have been edited at least once, and there have been more than 21.8 million post edits² (see Figure 3.1 for an example).

Studies [124, 125] show that post editing improves the content quality. By analyzing the Stack Overflow’s archival post edits, we find that some post editing involves complex revisions such as adding or removing sentences, code snippets or web resources. But there are also large numbers of post edits which involve small sentence revisions, such as correcting misspellings or grammar errors in a sentence, or changing the word or sentence format according to the website- or domain-specific convention. For example, Figure 1 shows a post edit with corrections of misspellings, grammar errors and formatting. Table 1 lists more examples of different kinds of small sentence edits. Among all 8.5 million post edits annotated with comments in Stack Overflow, 2.1 millions (24.7%)³ of them contain keywords like “spell”, “grammar” or “format” in the

¹<https://stackoverflow.com/help/editing>

²In Stack Overflow, edits by post owners or trusted contributors will be directly accepted. Edits by novice contributors will be accepted only if they are approved by three trusted contributors. Hereafter, post edits refer to accepted post edits, unless otherwise stated.

³This number could be highly underestimated as there are many similar words which are not taken into account, such as “typo”, “wording”, “indentation”, etc. See Figure 3.3.

comments of post edits (See Section 3.2.3). Our survey of trusted contributors⁴ in Stack Overflow who has made large numbers of post edits to their own or others' posts further confirms the core users' attention to small sentence edits to improve the content quality (See Section 3.6).

The presence of a large number of small sentence edits and the attention of trusted contributors to such edits motivate us to investigate a post edit assistance tool for identifying minor textual issues in posts and recommending proper corrections, such as those shown in Figure 3.1 and Table 3.1. Our analysis of "who edited posts" (see Section 3.2.2) reveals that 65.77% of all accepted post edits are self-edits by the post owners and 34.23% are collaborative edits by some post editors. Among collaborative edits, 89% are done by trusted contributors and 11% are done by novice contributors. Therefore, an edit assistance tool will not only help post owners reduce minor textual issues before posting their questions and answers, but also help post editors improve their editing efficiency. Furthermore, the identified issues together with the recommended corrections will help novice post editors learn community-adopted editing patterns.

To determine the required tool support, we conduct a formative study to understand the categories, editing reasons and scale of changes of post edits in Stack Overflow. As shown in Table 3.1, some sentence edits (row 1-5) is to fix general language issues, such as misspellings, grammar errors, sentence formatting issues. Some general language issues can be resolved by spell checking tools like LanguageTool⁵. However, due to the lack of domain knowledge, general spell checkers often make mistakes, such as changing *css* (acronym for Cascading Style Sheets) to *CBS*, *json* (acronym for JavaScript Object Notation) to *son*, or *li* (an HTML tag) to *Ali*. Furthermore, there are many sentence edits beyond general language issues, such as site-specific formatting style (row 6-8) and domain-specific naming convention (row 9-11). For example, when mentioning a User Interface (UI) component (e.g., the *OK* button), the community prefers to quote

⁴In Stack Overflow, users with 2000+ reputation scores are considered as trusted contributors. Other users are considered as novice contributors.

⁵<https://languagetool.org/>

the component name. When listing items, the community prefers to use Markdown language⁶ (e.g., HTML `<li>`). Software-specific names should be mentioned according to the domain-specific convention, like *AppleScript* rather than *Apple script*.

Considering the devisity of words and formats involved in sentence edits, it would require significant manual effort to develop and validate a complete set of rules for representing editing patterns. For example, OK should be quoted when it refers to an UI component, but it will not be quoted in many other contexts. It is impractical to enumerate all such cases.

Alternatively, machine learning techniques can be used to learn sentence editing patterns from archival post edits. In this work, we formulate sentence editing as a machine translation problem, in which an original sentence is “translated” into an edited sentence. We solve the problem using the RNN encoder-decoder model [35]. As we observe that the majority of post edits involves only character-level changes of post content, we decide to use a character-level RNN model. Furthermore, we develop a normalization method for preprocessing and postprocessing domain-specific rare words (e.g., URLs, API calls, variable names) in posts before and after the RNN model training. To train the RNN model, we develop a text differencing algorithm to collect a large dataset of original-edited sentence pairs from post edits, such as those shown in Table 3.1⁷. The trained RNN model detects and encodes editing patterns from original-edited sentence pairs, thus removing the need for prior manual editing rule development.

We evaluate the quality of sentence editing recommendations by our approach with large-scale archival post edits. Our approach outperforms the LanguageTool (a rule-based proof-reading tool that has been developed for over 10 years) and a phrase-based SMT technique [126] specially designed for sentence correction. We also conduct a field study in which the first author acts as a novice post editor that has little post editing experience. Based on the sentence editing recommendations by our tool, he edits 39 posts, 36 of which have been accepted. That is, for each accepted post edit, at least three trusted contributors consider that the edit has significantly improved the post quality.

⁶<http://stackoverflow.com/editing-help>

⁷Edits in Table 3.1 are underlined for the explanation purpose. The RNN model does not know such sentence edits in advance.

Finally, we collect 58 replies in a survey of trusted contributors who has made large numbers of post edits. These replies shed the light on the trusted contributors' opinions and suggestions for our tool.

We make the following contributions in this chapter:

- We conduct a formative study of post editing in Stack Overflow and a survey of trusted contributors' attention to small sentence edits, which help us understand the need for an edit assistance tool and who can benefit from the tool in which scenarios.
- We develop a RNN-based edit assistance tool for identifying spelling, grammar and formatting issues in Stack Overflow posts and recommending corrections. The tool learns editing patterns from big data of accepted archival post edits and makes special considerations of the data characteristics of post edits. To the best of our knowledge, our dataset is the largest dataset that has ever been collected for sentence editing task.
- We evaluate our approach from three perspectives, including the quality of sentence editing recommendations using large-scale archival post edits, the ability to assist a novice post editor in editing unfamiliar posts, and the feedbacks of 58 trusted contributors on our tool.

3.2 Formative Study of Post Editing in Stack Overflow

Stack Overflow is selected as our study subject, not only because of its popularity and large user base, but also because it is a well-known online Q&A site which supports collaborative editing [124, 125]. We download the latest data dump⁸ of Stack Overflow which contains 33,402,449 posts (including 12,865,526 questions and 20,536,823 answers) and all post edits since the launch of Stack Overflow in 2007 to December 11, 2016. Based on this large dataset, we carry out an empirical study of post edits in

⁸<https://archive.org/download/stackexchange>

FIGURE 3.2: The numbers of posts and three kinds of post edits in each year in Stack Overflow

Stack Overflow to understand the characteristics of post editing in Stack Overflow and to motivate the required tool support.

3.2.1 What has been edited?

In Stack Overflow, there are three kinds of post information which can be edited, i.e., question tags, question title, and post body [124]. Question-title and post-body editing are of the same nature (i.e., sentence editing), while question-tags editing is to add and/or remove the set of tags of a question.

As of December 11, 2016, there have been in total 21,759,565 post edits. Among them, 1,857,568 (9%) are question-title edits, 2,622,955 (12%) are question-tag edits, and the majority of post edits (17,279,042 (79%)) are post-body edits. The tags of 2,246,658 (17.5%) questions, the titles of 1,630,933 (12.7%) questions, and the body of 11,205,822 (33.5%) posts have been edited at least once. Figure 3.2 shows that the number of post edits increases as the number of posts increases over time. The number of question-title and question-tag edits increase slowly, while the number of post-body edits increase in a similar rate as posts increase.

Overall, post-body and question-title edits make up the majority of post edits. Compared with adding/removing question tags, post-body and question-title editing are more complex (further studied in the next question). Therefore, we focus on post-body and question-title edits in this work. Hereafter, post edits refer to post-body and question-title edits, unless otherwise stated.

FIGURE 3.3: The word cloud of the top 50 most frequent words appearing in edit comments (the words are aligned alphabetically from left to right)

3.2.2 Who edited posts?

Among all 19,136,610 post-body and question-title edits, 12,586,199 (65.77%) are self-edits by the post owners, 5,806,880 (30.34%) are collaborative edits by trusted contributors, and 743,531 (3.89%) are collaborative edits by novice contributors. This data suggests that an edit assistance tool may benefit the Stack Overflow community from three perspectives.

First, the tool can highlight the textual issues in the posts that the post owners are creating and remind them fixing the issues. This helps to get the posts right in the first place and reduce the need for after-creation editing. Second, trusted contributors make up only 9% of Stack Overflow users but they take up 30.34% of post editing tasks. An edit assistance tool that recommends sentence edits can improve the editing efficiency of trusted contributors. Third, the contribution by novice editors to post editing is rather low. This could be because novice editors do not have enough experience and courage to edit others' posts, or their edits are be incorrect and rejected by trusted contributors. An edit assistance tool's edit recommendations can serve as a "tutor" to teach novice contributors the community-adopted editing patterns. This may help to on-board novice contributors in Stack Overflow and improve the quality of their post edits.

3.2.3 What are post edits about?

According to the Stack Overflow's edit guidelines⁹, there are four common types of edits: 1) to fix grammatical or spelling mistakes, 2) to clarify the meaning of a post without changing it, 3) to correct minor mistakes or add addendums/updates as the post ages, 4) to add related resources or hyperlinks.

We analyze the comments of post edits to understand what post edits are about and whether they align with the community guideline. In Stack Overflow, when users finish editing a post, they can add a short comment to summarize the post edit. We collect all post-edit comments and apply standard text processing step to post-edit comment such as removing punctuation, lowercasing all characters, excluding stop words, stemming. Then we count the word frequencies and we display the top 50 most frequent words in a word cloud [127] in Figure 3.3. The font size of a word depends on the word frequency¹⁰ in our dataset.

According to these comments, it can be seen that post edits have covered four common edit types in the guideline, such as “spelling”, “typo”, “grammar” for the type (1), “clarification”, “details”, “explanation” for type (2), “fixes”, “errors”, “changes” for type (3), and “links”, “information”, “image” for type (4). Apart from them, there are also some other keywords, such as “formatting”, “indentation”, “highlighting”, “capitalization”, and “readability”. Although formatting, grammar and spelling types of edits are not about post mistakes or additional/updated resources, they are still crucial for readers as these edits can make the posts easier to read and understand. Table 3.1 lists some examples of formatting, grammar and spelling edits. In fact, the word cloud shows that formatting, grammar and spelling types of edits happen more frequently than other types of edits.

⁹<http://stackoverflow.com/help/privileges/edit>

¹⁰We normalize the frequency in logarithm to avoid the extreme large word size in the figure.

FIGURE 3.4: The count of original-edited post title and body in different similarity range

3.2.4 What is the scale of changes that post edits involve?

When editing a post, users may change some words, delete a sentence, add some sentences or code snippets according to different goals or context. To understand the scale of changes that post edits involve, we measure the similarity between the original post before a post edit and the edited post after the edit. Given a post edit, let *original* and *edited* be the text content (question title or post body) of the original and edited post. We use the text-matching algorithm [128] to find the character-level Longest Common Subsequence (LCS) between the *original* and *edited* content. We measure the similarity between the original and edited post as:

$$similarity(original, edited) = \frac{2 * N_{match}}{N_{total}} \quad (3.1)$$

where N_{match} is the number of chars in the LCS and the N_{total} is the total number of all chars in both the *original* and *edited* content. The similarity score is in the range of 0 to 1. The higher the similarity score, the less changes between the original and edited post.

As shown in Figure 3.4, among the 17,279,042 post-body edits, the original and edited post body of 55.28% edits are very similar with the similarity score between 0.9 and 1. 16.01% of them are between 0.8 to 0.9. Similarly, among 1,857,568 question-title edits, 64.47% of them are between 0.8 and 1. This indicates that most of the post edits involve only minor scale of changes of question titles and post bodies.

character-level changes of post edits like formatting, grammar, spelling, our approach trains a character-level RNN encoder-decoder model with a large parallel corpus of original-edited sentence pairs. The trained sentence editing model can identify minor textual issues (both general and domain-specific) in an original sentence and recommend corrections of these issues. Next, we will describe the core steps of our approach.

3.3.2 Collecting the corpus of original-edited sentence pairs

A post may have been edited several times. Assume a post has N versions, i.e., undergoing $N - 1$ post edits. For each post edit i ($1 \leq i \leq N - 1$), we collect a pair of the original and edited content. The original content is from the version i of the post before the edit, and the edited content is from the version $i + 1$ of the post after the edit. The edited part can be question title or post body. As this work focuses on sentence edits, we remove code snippets by HTML tags “< code >” from the collected content. Then,

we split the content into sentences by punctuation such as “.”, “?”, “!” and “;”.

ALGORITHM 1: Collect original-edited sentence pairs from post edits

Input: Two sentence lists $oList$ (original) and $eList$ (edited)

Output: A list of original-edited sentence pairs $pList$

Init $oIndex \leftarrow 0$, $eIndex \leftarrow 0$;

while $oIndex < oList.length$ **&&** $eIndex < eList.length$ **do**

 Init $largestScore \leftarrow -1$, $topPosition \leftarrow -1$;

for $i \in [eIndex, eList.length-1]$ **do**

if $oList[oIndex] == eList[i]$ **then**

$eIndex = i + 1$;

$largestScore = 1$;

break;

end

if $largestScore \neq 1$ **then**

for $i \in [eIndex, eList.length-1]$ **do**

$similarity = computeSimilarity(oList[oIndex], eList[i])$;

if $similarity > largestScore$ **then**

$largestScore = similarity$;

$topPosition = i$;

end

if $largestScore > sim_threshold$ **then**

$pList.append([oList[oIndex], eList[topPosition]])$;

$eIndex = topPosition + 1$;

$oIndex = oIndex + 1$;

end

The Algorithm 1 aligns the sentence list $oList$ from the original content and the sentence list $eList$ from the edited content. It finds the LCS of matched (lines 4-10) and unmatched-but-similar-enough (lines 11-23) sentences between the two input sentence lists. For the two unmatched sentences, $computeSimilarity()$ computes the char-level LCS of the two sentences [128] and measures the sentence similarity using the Eq. 3.1. For a sentence in the $oList$, if the similarity score $largestScore$ of the most similar sentence in the $eList$ is above a threshold $sim_threshold$, the two sentences are aligned as a pair of original-edited sentences. Similarity threshold should be set to achieve

FIGURE 3.6: Basic RNN model

a balanced precision and recall for sentence alignment, so we experimentally set the threshold at 0.8 in this work. The algorithm outputs all aligned original-edited sentence pairs.

From the post edits before Dec 11, 2016, we collect 13,806,188 sentence pairs. But there are two common problematic kinds of sentence pairs in the dataset. First, some sentence pairs are code snippets which are not enclosed inside `< code >` HTML tag. Such code-snippet sentences are not in the scope of our study. We exclude code-snippet sentences if sentences contain programming constructs such as “{”, “}”, “=”, “if(”, “for(”, “while(”. Second, sometimes a post is edited by one user, but then is edited back into its original content by another user. We cannot decide which one of the edits is more suitable. Therefore, we exclude such sentence pairs. After post-processing, 7,545,979 sentence pairs are left, which are used to train the RNN encoder-decoder model for automatic sentence editing.

3.3.3 Character-level RNN Encoder-Decoder Model

Recurrent Neural Network (RNN) is a class of neural networks where connections between units form directed cycles. Due to its nature, it is especially useful for tasks involving sequential inputs such as speech recognition [129] and sentence completion [33]. Compared with traditional n-gram language model [130], a RNN-based language model can predict a next word by preceding words with variant distance rather than a fixed number of words. This makes it possible to model long-distance dependencies in the sentence.

The architecture of a basic RNN model includes three layers. An input layer maps each word to a vector using word embedding or one-hot word representation. A recurrent

FIGURE 3.7: The RNN encoder-decoder model

hidden layer recurrently computes and updates a hidden state after reading each word. An output layer estimates the probabilities of the next word given the current hidden state. Figure 3.6 shows the unfolding in time of the RNN's forward computation. At time step t , it estimates the probability of the next word $P(w_{t+1} | w_1, \dots, w_t)$ by three steps. First, the current word w_t is mapped to a vector x_t by the input layer.

$$x_t = \text{input}(w_t) \quad (3.2)$$

Then, the hidden layer generates the hidden state h_t according to the previous hidden state h_{t-1} and the current input x_t

$$h_t = h_{t-1} * W + x_t * U \quad (3.3)$$

where W , U are parameters inside the neural network. Finally, the $P(w_{t+1} | w_1, \dots, w_t)$ is predicted according to the current hidden state h_t :

$$P(w_{t+1} | w_1, \dots, w_t) = g(h_t) \quad (3.4)$$

where the function g produces valid probabilities. During model training, the parameters are learned by backpropagation [131] with gradient descent to minimize the error rate.

More complex RNN-based models have been developed for more complex NLP tasks. For example, the RNN encoder-decoder model [35] is commonly adopted for machine translation tasks. This model includes two RNNs as its main components: one RNN to encode a variable-length sequence into a fixed-length vector representation, and the other RNN to decode the given fixed-length vector representation into a variable-length sequence. From a probabilistic perspective, this model is a general method to learn

the conditional distribution over a variable-length sequence conditioned on yet another variable-length sequence, i.e., $p(y_1, \dots, y_{T'} \mid x_1, \dots, x_T)$. The length of the input T and output T' may differ.

The architecture of our character-level RNN encoder-decoder model is shown in Figure 3.7. The example is to edit “is java oo.” to “Is Java OO?” in which “OO” is the abbreviation of “Object Oriented”. The encoder is a basic RNN model that reads each character of an original sentence x sequentially. This work focuses on sentence edits that involve many character-level changes such as misspellings, capitalizations, annotations. Furthermore, the character-level model can avoid the out-of-vocabulary problem [132, 133] because there are countless words, but only limited characters. Therefore, we use the character-level RNN model instead of the normal word-level one. As the model reads each character sequentially, the hidden state of the RNN encoder is updated according to Eq. 3.3. After reading the end of the the input, the hidden state of the RNN encoder is a vector c summarizing the whole input original sentence.

The decoder of the model is another RNN which is trained to generate the output edited sentence by predicting the next word y_t given the hidden state $h_{(t)}$. Unlike the basic RNN model in Figure 3.6, y_t and h_t are not only conditioned on y_{t-1} but also on the summary vector c of the input sentence. Hence, the hidden state of the decoder at time t is computed:

$$h_t = f(h_{t-1}, y_{t-1}, \mathbf{c}) \quad (3.5)$$

and the conditional distribution of the next character is

$$P(y_t \mid (w_1, \dots, w_{t-1}), \mathbf{c}) = g(h_t, y_{t-1}, \mathbf{c}) \quad (3.6)$$

for the given activation functions f and g . The two RNN components of the RNN encoder-decoder model are jointly trained to maximize the conditional log-likelihood

$$\max_{\theta} \frac{1}{N} \sum_{n=1}^N \log p_{\theta}(y_n \mid x_n) \quad (3.7)$$

where θ is the set of the model parameters and each (x_n, y_n) is a pair of original-edited sentences from the training corpus.

Original: pls check <https://docs.python.org/2/library/itertools.html#itertools.groupby> for `itertools.groupby()` ...

Preprocess: pls check UNK_URL₁ for UNK_API₁...

Edited: Please check UNK_URL₁ for UNK_API₁...

Post-process: Please check <https://docs.python.org/2/library/itertools.html#itertools.groupby> for `itertools.groupby()` ...

FIGURE 3.8: The process to deal with domain-specific rare words

3.3.4 Normalizing domain-specific rare words

The performance of deep learning models heavily depends on the quality of the training data. In particular, our RNN-based model relies on patterns of character sequences. However, in domain-specific Q&A text like Stack Overflow discussions, many terms are problem-specific or external resources, such as URLs of online documents (e.g., <http://support.microsoft.com/kb/299853>), API calls (e.g., `document.getElementById('whatever')`) and variable names (e.g., `pages[0]`). According to our observation, these specific terms usually have few errors because developers are more careful and sensitive when mentioning API/variable names than other general text. For the URL links, most of them are put into the text by copy-paste which rarely lead to errors.

However, when dealing with such special character sequences, the RNN encoder-decoder model cannot learn meaningful patterns effectively due to their rareness and diversity in text. Furthermore, these problem-specific and resource terms will negatively influence the quality of sentence editing model because they introduce noise to other normal words. Therefore, we normalize mentions of such domain-specific rare words in the training sentences to simplify the model complexity.

The normalization process first detects mentions of domain-specific rare words by regular expressions. Each detected mention in the original and edited sentence will be marked by a unique special symbol. For Stack Overflow sentences, we develop regular expressions for detecting URLs by `http://` or `https://`, API calls by API conventions like `a.b(c)` (Java/Python) or `a::b(c)` (C/C++), and array/list variables by `a[]`. As the example in the Figure 3.8 shows, <https://docs.python.org/2/library/itertools.html#itertools.groupby> and `itertools.groupby()` are marked as UNK_URL₁ and UNK_API₁ after normalizing the training original-edited sentences. UNK_URL_{index}, UNK_API_{index}, or UNK_VARIABLE_{index} are

special symbols where *index* is the unique index of an URL/API/variable in the training corpus.

The normalized sentence pairs are used to train the RNN encoder-decoder model. Given an original sentence to be edited, the trained model will change it into an edited sentence. The special symbols are then mapped back to the original domain-specific words in a post-processing step.

3.3.5 Implementation

Slightly different from the model in Figure 3.7, our implementation of the RNN encoder-decoder model consists of 3 hidden layers, i.e., deeper model structure for better learning. In each hidden layer, there are 1024 hidden units to store the hidden states. We implement our model based on the Tensorflow [134] framework and train the model in a Nvidia M40 GPU (24G memory) for about 6 days.

3.4 The Quality of Recommended Sentence Edits

Our edit assistance tool aims to help post owners and editors edit posts by identifying textual issues in sentences and recommending small sentence edits for correcting these issues. The quality of recommended sentence edits will affect the adoption of the tool by the community. In this section, we use randomly selected 377,298 original-edited sentence pairs from archival post edits to evaluate the quality of recommended sentence edits by our tool.

3.4.1 Dataset

From the accepted 19,136,610 post edits, we collect 7,545,979 original-edited sentence pairs. We randomly take 6,791,381 (90%) of these sentence pairs as the training data, 377,298 (5%) as the development data to tune model hyperparameters, and 377,298

(5%) as the testing data to evaluate the quality of recommended sentence edits by our tool.

3.4.2 Baseline methods

Apart from our own model, we take another two methods as baselines for comparison. One baseline is the LanguageTool¹¹, an open source proof-reading program for more than 20 languages. This tool’s style and grammar checker is rule-based and has been developed for over 10 years. The other baseline is the phrase-based SMT model specifically designed for sentence correction [126]. We use the same training data to train the SMT model.

3.4.3 Evaluation metric

Our task can be regarded as a domain-specific GEC task, as it deals with the site- and domain-specific formatting, grammar and spelling knowledge. Therefore, we adopt GEC metrics for evaluating our approach.

Precision and recall have been traditionally used to evaluate the performance of GEC approaches [36, 135]. Given a sentence, precision measures the percentage of edits suggested by a tool that are correct, and recall measures the percentage of correct edits that are suggested by the tool. Precision and recall require manually-annotated gold-standard edits, such as insert, deletion, replacement, tense change, etc., in the sentences [37]. For example, the underlined text in the reference sentences in Table 3.2 are manually annotated gold-standard changes for the corresponding original sentences. However, the difficulty of defining error types and the disagreement between annotators often challenge the annotation validity as a gold standard [136]. This is especially the case for our data, considering the large number of techniques and concepts that have been discussed in Stack Overflow and the varieties of sentence edits that have been applied.

¹¹<https://languagetool.org/>

Original Sentence	how do i make a file handle from a file path specified on the command line?	Would you recommend Java/J2EE, .Net / Erlang ?
Edits by a tool	How do I make a file handle from a file path specified on the command line?	Would you recommend Java/J2EE, .NET / Erlang ?
Reference	How do I make a <u>filehandle</u> from a file path specified on the command line?	Would you recommend Java/J2EE, .NET or Erlang?
GLEU score	48.36	27.08

TABLE 3.2: GLEU score for different edits

In the GEC field, recent released shared tasks have prompted the development of GLEU [137, 138] (Generalized Language Understanding Evaluation¹²) for evaluating GEC approaches. GLEU is a customized metric from the BLEU (BiLingual Evaluation Understudy) [139] score which is widely used to evaluate the machine-translation quality. It is independent of manual-annotation scheme and requires only reference sentences (without annotations of gold-standard edits). Recent studies [136, 140] show that GLEU has the strongest correlation with human judgments of GEC quality and effort, compared with precision and recall.

Therefore, we adopt GLEU in our evaluation. We regard an original sentence as the source sentence (S), the edited sentence by Stack Overflow user as the reference sentence (R), and the edited sentence generated by a GEC tool as candidate sentence (C). GLEU score measures how close a candidate sentence generated by the tool is to the reference sentence edited by human, with respect to the source sentence. Intuitively, GLEU awards the overlap between C and R but not in S , and penalizes n-grams in S that should have been changed but are not and n-grams found in S and C but not in R . It also captures the sentence length and the fluency and adequacy of n-gram overlap. GLEU is computed as:

$$GLEU(S, R, C) = BP \cdot \exp\left(\sum_{n=1}^N w_n \log p_n\right) \quad (3.8)$$

where p_n is the number of n-gram matches between the candidate sentence (C) and the reference sentence (R), minus the sum of positive difference of n-gram matches between candidate-source sentences (C, S) and candidate-reference sentences (C, R), divided by the number of n-grams in the candidate sentence (C):

$$p_n = \frac{\sum_{ng \in \{C \cap R\}} count_{C,R}(ng) - \sum_{ng \in \{C \cap S\}} \max[0, count_{C,S}(ng) - count_{C,R}(ng)]}{\sum_{ng \in C} count(ng)} \quad (3.9)$$

¹²<https://github.com/cnap/gec-ranking>

Method	GLEU
RNN encoder-decoder	57.44
LanguageTools	51.93
SMT	46.85

TABLE 3.3: The performance of different methods for sentence editing

Original Sentence	Our RNN Encoder-Decoder	LanguageTool	Phrase-based SMT
1 What I did wrong?	What <u>did I do</u> wrong?	What I did wrong?	What I did wrong?
2 pls help me..	<u>Please</u> help me.	<u>Pls</u> help me.	<u>Please</u> help me.
3 However, you show use CSS for this.	However, you <u>should</u> use CSS for this.	However, you show use <u>CBS</u> for this.	However, you use for this.
4 I'm thinking it has something to do with the json.	I'm thinking it has something to do with the <u>JSON</u> .	I'm thinking it has something to do with the <u>gan</u> .	I'm thinking it has something to do with the <u>JSON</u> .
5 Inside the li tag we have many options to select.	Inside the <u>li</u> tag we have many options to select.	Inside the <u>Ali</u> tag we have many options to select.	Inside the tag we have many options to select.
6 Here selectedShape is either circle or polygon.	Here "selectedShape" is either circle or polygon.	Here selectedShape is either circle or polygon.	Here selectedShape is circle or polygon.
7 Edit: By the way, this is my first time using the community wiki.	<u>**Edit**</u> By the way, this is my first time using the community wiki.	Edit: By the way, this is my first time using the community wiki.	Edit: By the way, this is my first time using the wiki.
8 It looks s if the Large image is taking up all the space.	It looks <u>as</u> if the Large image is taking up all the space.	It looks <u>as</u> s if the Large image is taking up all the space.	It looks <u>as</u> if the <u>**Large**</u> is taking up all the space.
9 Below the code I use to validate a user login outside magento.	Below the code I use to validate a user login outside magento.	Below the code i use to validate a user logic outside <u>magenta</u> .	the code I use to validate user login outside <u>magento2</u> .
10 How to find hte library dependency?	How to find <u>the</u> library dependency?	How to find <u>Rix</u> library dependency?	How to find <u>the</u> library dependency _a ?

TABLE 3.4: Examples of sentence edits by different methods

where $ng \in \{A \cap B\}$ are common n-grams in sentence A and B , and $count_{A,B}(ng)$ is the minimum occurrence number of n-grams in A and B . $n = 1, \dots, N$ where N is the maximum number of grams to be considered. w_n is the weight of p_n . We set N to 4 which is a common practice in the machine translation and grammatical error correction literature, and set all $w_n = \frac{1}{N}$. BP is a brevity penalty which penalizes short candidate sentence (that may have a higher p_n due to the small number of n-grams in the sentence).

$$BP = \begin{cases} 1 & c > r \\ e^{(1-\frac{r}{c})} & c \leq r \end{cases} \quad (3.10)$$

where r is the length of the reference sentence, and c is the length of the candidate sentence.

GLEU is expressed a percentage value between 0 and 100. The higher the GLEU, the closer the candidate sentence is to the reference sentence. If the candidate sentence completely matches the reference sentence, the GLEU is 100.

3.4.4 Evaluation results

We report out evaluation results by answering the following two research questions.

3.4.4.1 What is the quality of recommended sentence edits by our method? How much improvement can it achieve over the baseline methods?

Table 3.3 presents the GLEU score of different methods for sentence editing tasks in the testing dataset of 377,298 sentences. Our RNN encoder-decoder model achieves the best overall result with the average GLEU score 57.44. The average GLEU of the SMT is only 46.85, and the average GLEU of the LanguageTool is 51.93. According to the literature of machine translation [34, 141] and grammar error correction [30, 31], the improvement in the GLEU score by our model represents a significant improvement over the two baseline methods.

GLEU score is a relative strict evaluation metric, i.e., any editing mistake may lead to large decay of the GLEU score. Consider the original sentence “how do i make a file handle from a file path specified on the command line?” in Table 3.2. There should be three edits: *how* to *How*, *i* to *I*, and *file handle* to *filehandle*, as seen in the reference sentence. Our RNN model suggests the first two edits, but misses the third edit. But the GLEU score is only 48.36. Similarly, for the other sentence “Would you recommend Java/J2EE, .Net/Erlang?”, our RNN model suggests one edit, but miss the other edit. The GLEU score is only 27.08. However, compared with precision (100% in both examples) and recall (66.7% and 50% respectively), GLEU can better reflect editing quality with respect to editing effort required.

To qualitatively understand the strengths and weakness of different methods, we randomly sample about 600 sentences for manual inspection. Table 3.4 lists some representative categories of examples in which our method outperforms the two baseline methods. Due to the page limitation, other examples (both high-quality and low-quality) can be found online¹³. We can see that compared with the two baseline methods, our model can carry out relatively complex edits (e.g., the first example) and domain-specific word and site-specific formatting (e.g., the 4th, 5th, 6th and 7th examples). In such cases, the LanguageTool mostly preserves the original sentences because it does not have rules for them. Even worse, because many domain-specific words (e.g., *css*, *json*, *magento*) are

¹³http://tagreorder.appspot.com/sentenceCorrection_examples.html

out of its vocabulary, the LanguageTool may regarded them as typos of some general words and make erroneous edits, such as *CBS* for *css* (the 3rd example), *son* for *json* (the 4th example), *magenta* for *magento* (the 9th example).

The SMT can edit some domain-specific words (e.g., *json* to *JSON* in the 4th example). But it often preserves the original sentences that should be edited (e.g., the 1st and 7th examples), removes words that should not be removed (e.g., the 3rd, 5th and 6th examples), formats the sentence (e.g., the 8th example) that should not be formatted, or introduces some strange words (e.g., the 9th example). In general, the phrase-based SMT does not work very well for minor scale of changes involved in post edits. Therefore, it often has worse GLEU than the LanguageTool.

3.4.4.2 In which cases does our model recommend low-quality sentence edits?

By analyzing low-quality recommendations by our tool, we find four main cases in which our model does not perform well.

First, some sentences are edited to add more information which is beyond the context of a sentence, such as editing “I am currently working on a large project that heavily uses smart pointers with reference counting.” to “I am currently working on a large project in C++ that heavily uses smart pointers with reference counting.”. Our current model considers only the local context of a sentence. To support such complicated edits, the broader context of the sentence (i.e., previous and subsequent sentences) need to be considered in the future.

Second, sometimes the context captured by our model may not be long enough. For example, the original sentence “My db engine is MySQL I have two table 1.” should be edited to “My DB engine is MySQL I have two table 1.”. But our method recommends not only capitalizing “db” to “DB”, but also changing “table” to “tables”. However, the LanguageTool will not make such a mistake because there are no rules inside it to change singular form to plural form.

Third, different users may have different opinions regarding what should or should not be edited. For example, some users will edit the sentence “Would you recommend Java/J2EE, .Net /

FIGURE 3.9: The performance of different methods dealing with sentences of different length (the number of characters of the sentence)

Erlang ?” to “Would you recommend Java/J2EE, .NET or Erlang?” by changing “/” to “or”. However, many other users will not do that. Many revert-back edits we see when collecting original-edited sentences are the evidence of such different opinions. Different editing opinions often result in non-obvious editing patterns, which machine learning techniques cannot effectively encode.

Fourth, the sentence length is a crucial factor influencing the performance of the RNN model. As shown in Figure 3.9, with the increase of the sentence length, the GLEU of all three methods becomes higher. This does not simply mean that these methods work better on longer sentences than shorter sentence. Instead, this is because GLEU favors longer sentences as they would require more effort to read and edit. For example, although the tool misses one edit for both sentences in Table 3.2, the GLEU for the longer sentence is much higher than the shorter sentence. For all sentence lengths we experiment, our model performs the best. But the performance difference between our model and the baseline methods narrows as the sentence length increases. The fact that GLEU favors longer sentences and the performance difference narrows actually indicates that the performance of our RNN model decays as the sentence length increases. That is because the RNN model may “forget” some prior information when processing towards the end of a long sentence. As such, it cannot encode long-distance editing patterns very well which will lead to edit mistakes.

3.5 Assisting Novice Post Editors in Post Editing

Having established the confidence in the quality of sentence edits by our tool, we would like to further investigate whether the recommended sentence edits by our tool can assist novice post editors who have little experience in post editing and have little expertise in post content in successfully completing post editing tasks.

To that end, we conduct a small-scale field study, in which the first author who has 400+ reputation score in Stack Overflow acting as a novice post editor. We randomly select 50 posts from April 4 to April 6, 2017 which is of reasonable size, and the human effort to manually collect the posts and submit the post edits is manageable.. These 50 posts are not in our dataset of archival post edits. In stack Overflow, each question can have up to 5 tags to describe its topics. The 50 selected posts contain in total 123 tags (if the post is the answer, we take tags from its corresponding question) and 105 of these tags are unique. This indicates that the 50 selected posts cover a diverse set of programming topics. In fact, these posts contain many technical terms that are beyond the expertise of the first author.

In Stack Overflow, novice post editors have to submit their post edits for peer review. According to the Stack Overflow policy¹⁴, to guarantee the quality of post edits, a post edit will be accepted only if at least three trusted contributors approve the edit otherwise it will be rejected. The first author uses our model to generate sentence editing recommendations for the selected 50 posts, based on which he edits the posts and submit the post edits to the community for approval. Our field study lasts for three days because each user in Stack Overflow can submit at most 5 post edits at the same time. We cannot submit more post edits until some submitted post edits are accepted or rejected.

Among the 50 selected posts, our model finds that 39 posts need at least one sentence edit and suggests the needed edits. For these 39 posts, there are on average 3.9 sentences edited and 5.6 edits per post (one sentence may receive several edits). Among the 39 submitted post edits, 36 (92.3%) are accepted and 3 (7.7%) are rejected. Records

¹⁴<https://meta.stackexchange.com/questions/76284>

1d	suggested	approved edit on	How do I unmount a div generated by a loop using id's in React.js?
1d	suggested	rejected edit on	How to compress image size in javascript
1d	suggested	approved edit on	How to make tab with Ajax and bootstrap
1d	suggested	approved edit on	Can not search for email using an index

FIGURE 3.10: Part of the results of post edits by our model

of some accepted and rejected edits¹⁵ are shown in Figure 3.10. 3 rejected post edits contain 1, 2, 4 sentence edits respectively. For example, one post edit involves only adding a question mark to the title, and it got three reject votes. The other two rejected post edits actually got two approval votes but one reject vote. Although our tool recommends the correct sentence edit, the post edit is rejected because it is regarded as too minor which does not significantly improve the post quality¹⁶. In other words, for the 36 accepted post edits, at least trusted contributors believe that they contain sufficient edits that significantly improve the post quality, and thus approve them.

3.6 Feedbacks of Trusted Post Editors

Finally, we would like to understand the trusted post editors' attention to small sentence edits and collect their opinions and suggestions for our edit assistance tool.

We design a survey with three 5-points likert scale questions and 1 free-text question. The likert scale questions are: 1) How much do you care about spelling, grammar, formatting edits? (1 being very little and 5 being very much); 2) What percentage of your edits are spelling, grammar, formatting edits? (1 being very low and 5 being very high); 3) How much could our tool help with such edits? (1 being very little and 5 being very much). The first and third questions are accompanied with the examples in Table 3.1 and Table 3.4 for illustration purpose. The free-text question asks for "any suggestions or opinions for our tool and this research".

To find survey participants, we sort all Stack Overflow users by the number of post edits they have made in descending order. We read the profiles of the top 2000 ranked users

¹⁵We cannot release the full results now as they will expose our identity which violates the double-blind policy.

¹⁶Predicting whether a post edit contains significant enough sentence edits is out of scope of this work.

and find the email contacts for 410 users. Each of these 410 users has at least 800 post edits to their own or others' posts. We send these 410 users an email introducing the background and goal of our work and providing access to the survey. Among these 410 users, we collect 61 valid survey replies¹⁷.

Figure 3.11 summarizes the results of the three likert scale questions. 55 of 61 survey respondents care much or very much about spelling, grammar, formatting edits. 30 respondents report that high or very high percentage of their edits is spelling, grammar, formatting edits. These results confirm the trusted post editors' attention to small sentence edits. 34 respondents consider that our tool would be helpful or very helpful for assisting small sentence edits.

34 of 61 respondents provides their opinions or suggestions for the free-text question. Those considering our tool helpful comment that "having a natural language and correct grammar is important as all accepted answers will be archived for reference in future", "SO needs this tool and I hope to see it in action soon. I believe the resulting tool might be useful outside the context of SO websites", and "Very good idea, that would deserve to be integrated into StackOverflow as an assistance tool". Some mention "make your tool free software (open source)".

However, even strong supporters have concerns like "How will it get integrated with the SO site?". Similar concerns are mentioned by those holding neutral opinion "I believe a tool like that would only be pretty useful if it somehow did what it did without at all getting in the way", and those considering our tool not helpful "dubious if you can create an interface that is sometimes useful and doesn't get in the way when it isn't".

Some respondents consider our tool not helpful because they do not like spell checking tools at all "Not interested. Same reason I abhor spell and grammar checkers. Generally way too many false positives". Others prefer to use existing tools "I use a browser plugin (Pentadactyl) to do the edits in an external editor (Vim), ... Any browser-based tool therefore would be of little value to me.". It would be interesting to see if these respondents would appreciate the uniqueness and quality of our tool if they had actually use the tool.

Finally, both consider-helpful and consider-not-helpful respondents suggest that we should consider assisting code formatting "Another useful thing would be to detect when some code isn't indented enough", "Most of my edits are fixing indentation and formatting to make the code more readable", "I don't care about

¹⁷Available at <http://tagreorder.appspot.com/surveyResults.html>

FIGURE 3.11: The survey results

English formatting. Code formatting is very important for me”. We believe they point out an interesting future direction.

3.7 Discussion

3.7.1 Unintended consequences of the edit assistance tool

Although the edit assistance tool seems to contribute to the community, it may also cause some unintended consequences. First, conflicts may occur when people’s questions and answers are edited by others¹⁸ even though those editors have good intentions. The wrong edits changing the post meaning may discourage the users actively engaging with the community. Although such situations exist, Li et al [124] demonstrate that most collaborative edits enhance the quality of the posts by attracting more votes. In addition, the edits from our assistance tool are limited to the misspellings, grammar and formatting errors which is far from the altering the post meaning. And our tool only provides the edit recommendations, and the user will have a double check and make the final decision which avoid the potential mistakes.

Second, some new users may rely on editing others’ post to familiarize the community and earn the reputation, while the edit assistance tool harm their participation to the community by closing the gate. However, according to our observation, only 3.89% edits are proposed by the novice users which takes up about 91% total users in Stack

¹⁸<https://meta.stackexchange.com/questions/28005>

Overflow. Such a small proportion indicates that the post edit task does not play an important role in “on boarding” new members. Furthermore, the edit assistance tool can help collaborative editing while can never replace the manual collaborative edits, because it can never correct a totally new word without seeing them before. In contrast, our edit assistance tool can learn the crowd intelligence from the history while some new and creative spelling or formating edits still needs manual efforts. So, the door to the manual editing is still open to the novice users.

3.7.2 Limitations

Our work is also subject to limitations. First, one important contribution of this research is to adopt deep-learning based machine translation model to assist collaborative edits in Stack Overflow. We would expect our finding to apply to different Q&A or crowd-sourcing systems which owns enough edit history as the training data. But further comparative studies are needed in order to confirm the extent to which these findings are generalizable.

The second limitation is that it is difficult to quantitatively measure how much effort can be saved by using our edit assistance tool. Although we can ask some novice users to carry out the user study to check their edit efficiency with or without our tool, they only take about 4% of all edits, according to our statistics. In contrast, most edits (about $\frac{2}{3}$) are carried out by the post owners and the other 30% are by the core users. It is difficult to mimic the situation when they are asking the real question with our assistance tool or find core users (After all, only 63,395 in this world but few around us). We try to measure the approximate our tool impact by counting 7.5 millions small sentence edits assuming that they can also be avoided by using our assistance tool, though such assumption is too strong. We will keep on thinking about it and leave it for our future research.

3.8 Conclusion

In this chapter, we investigate the need for and the feasibility of assisting small sentence editing for software-specific text like Stack Overflow. Our empirical observation of post edits in Stack Overflow and the survey of trusted contributors confirms the need for an edit assistance tool and the potential benefits for the community. It can also benefits the content searching in Stack Overflow. A deep learning based approach has been developed to learn to apply sentence editing patterns from large-scale post edits by the community. Our evaluation through large-scale archival post edits demonstrates the quality of recommended sentence edits by our tool.

Chapter 4

Mining Knowledge Graph from Stack Overflow to Understand Technology Landscape

4.1 Introduction

Although Chapter 3 [1] have proposed novel methods for supporting document search in Stack Overflow, it still require developers to read tens of different posts to read, understand, summarize. Such a process is time-consuming. In addition, some knowledge in Stack Overflow are implicit which cannot be directly searched. In this chapter, we are trying to build a software-specific knowledge graph by mining tags in Stack Overflow. Compared with the document search in last two chapters, the entity search in our knowledge graph render direct answers to developers' queries.

A diverse set of technologies are available for use by developers and such set continues to grow. In this chapter, we use the term “technology” to broadly refer to processes, methods, tools, platforms, languages, and libraries in the context of software engineering. To make the right choice for a technology in a software project, developers need to have a good understanding of the technology landscape, i.e., available technologies, the relationships among them, and the trends of them. To that end, developers often

FIGURE 4.1: Post #1710476 in Stack Overflow

turn to two information sources on the Web [142]. First, domain experts often write articles about technology landscape, such as *“best machine learning resources for getting started”*, *“Python’s SQLAlchemy vs other ORMs”*, *“20 best JavaScript charting libraries”*, Second, developers can seek answers from community-curated list of useful technologies (e.g., *“awesome PHP”*) or from Q&A websites such as Stack Overflow or Quora (e.g., *“which framework is best for web development in PHP?”*). These expert articles and community answers are indexable by search engines, thus enabling developers to find answers to their technology landscape inquiries.

However, there are three limitations with these expert articles and community answers. First, the technology landscape is in a constant state of change. Thus, expert articles and community answers are easily out of date. For example, the article *evaluation of .net mocking libraries* compares “how the playing field looks today” (as of December 14, 2013) and “how the playing field looked two years ago”. It is in the top 10 list that Google returns for “best .net mocking framework”, but it cannot reflect the state-of-the-practice “today” (as of January 2016). Second, expert articles and community answers usually focus on a specific technology, while not a set of correlated technologies. For example, reading the article *“best PHP framework for 2015”*, one cannot know that *Symfony* uses a separate ORM library *Doctrine*, while *Laravel* includes a built-in ORM library *Eloquent*. A developer would need to read another article like *“best available PHP ORM libraries”* to aggregate the information. Such information aggregation is opportunistic. Third, expert articles and community answers are often primarily opinion-based. This is why Stack Overflow usually closes technology-landscape-style questions (e.g., *“C# - Which Unit Testing Framework”*), because such questions often solicit debate and arguments.

Several empirical studies [8, 9, 54] show that taken in the aggregate, Stack Overflow question tags provide a good estimation of technology landscape over time. Figure 6.6

shows an example. We can see that the tags identify the main technologies that the question revolves around, even those that are not explicit in the question content (in this example *java* and *orm* (object-relational-mapping)). Technologies that tags represent are correlated. In this example, *hibernate* is an *orm* framework for accessing a *sql* database from a *java* program. However, as Stack Overflow manages the question tags as a set of words, the relationships among tags and the tag usage over time are implicit in the system.

In this chapter, we propose to apply association rule mining [143] and community detection [144] techniques to mine the technology landscape from Stack Overflow question tags. The mined technology landscape is represented as a graphical Technology Associative Network (TAN) which can be regarded as a naive version of the knowledge graph but without explicit edge information. For each tag in the TAN, we use the Natural Language Processing (NLP) method [100] to analyze the tag description to determine if the tag represents a software library, programming language, or general concept. We also summarize the question asking activities of the tag over time.

We apply our approach to Stack Overflow data dump¹ and evaluate the mined TAN from the perspectives of tag and question coverage, semantic distance of technology associations, network structure, and network evolution. Our evaluation shows that the mined TAN captures a wide range of technologies, the complex relationships among technologies, and the trends of technologies. We release the mined TAN in our website. The website supports some basic search and exploration features. The Google Analytics results of the website usage data for about 4 months provides initial evidence of the public interests in the technology landscape². A small-scale user study is conducted, which demonstrates the potentials of the mined technology landscape in assisting technology search and exploration.

We make the following contributions in this chapter:

- a systematic approach for mining and analyzing TAN (naive knowledge graph) from Stack Overflow;

¹<https://archive.org/details/stackexchange>

²The website serves mainly as a web portal (prototype) to demonstrate our empirical results.

FIGURE 4.2: The general TAN mined at minimal support 0.0007 and minimal confidence 0.15

- a foundational study of semantic, structural and dynamic properties of the mined technology landscape;
- a web site <https://graphofknowledge.appspot.com/> and a browser plugin https://github.com/tomhanlei/kg_plugin for the public access of our technology landscape service;
- a user study for evaluating the usefulness of the mined technology landscape.

4.2 The Approach

In this work, we focus on how we can obtain a technology landscape like those shown in Figure 4.2. Manually creating a technology landscape of tens of thousands of technologies obviously would require significant time and human efforts. In this section, we introduce our approach to automatically mine technology landscape from Stack Overflow question tags. Our approach leverages the fact that structured knowledge of technologies can emerge from the tagging practices of millions of Stack Overflow users taken together [55, 57].

4.2.1 Mining technology associations

In this work, we consider Stack Overflow question tags as *technologies* for computer programming. Given a set of Stack Overflow questions, we use association rule mining [143] to mine technology associations from tag co-occurrences in questions. If the input set of questions contains all the Stack Overflow questions, we refer to the resulting TAN as the general TAN. If the input set of questions contains only questions that are tagged with some technologies, we refer to the resulting TAN as technology-specific TAN. If the input set of questions contains questions that are asked during a period of time, we refer to the resulting TAN as time-specific TAN.

In this work, a Stack Overflow question is considered as a transaction and the question tags as items in the transaction. As we are interested in constructing a TAN, we need to find frequent pairs of technologies, i.e., frequent itemsets that consist of two tags. A pair of tags is frequent if the percentage of how many questions are tagged with this pair of tags compared with all the questions is above the minimal support threshold t_{sup} .

$$support(t_i, t_j) = \frac{\#tagSet \text{ containing } (t_i \text{ and } t_j)}{size(reducedTagSets_{targetTerm})}$$

Given a frequent pair of tags $\{t_1, t_2\}$, association rule mining generates an association rule $t_1 \Rightarrow t_2$ if the confidence of the rule is above the minimal confidence threshold t_{conf} .

$$confidence(t_i \Rightarrow t_j) = \frac{\#tagSet \text{ containing } (t_i \text{ and } t_j)}{\#tagSet \text{ containing } t_i}$$

The confidence of the rule $t_1 \Rightarrow t_2$ is computed as the percentage of how many questions are tagged with the pair of tags compared with the questions that are tagged with the antecedent tag t_1 . With both support value and confidence value larger than the threshold, we obtain an association pair $t_i \Rightarrow t_j$. Note that our association rules involve only single item in antecedent and consequent

Given the mined tag association rules, we construct a TAN. A TAN is an undirected graph $G(V, E)$, where the node set V contains the tags (i.e., technologies) appearing in the association rules, and the edge set E contains undirected edges $\langle t_1, t_2 \rangle$ (i.e.,

technology associations) if the two tags has the association $t_1 \Rightarrow t_2$ or $t_2 \Rightarrow t_1$ ³. Each edge has a confidence attribute indicating the strength of the technology association.

4.2.2 Detecting technology communities

A TAN can consist of large numbers of technologies and the associations among technologies. Some relevant technologies would be strongly connected to each other, but loosely connected to those irrelevant technologies. In graph theory, a set of highly correlated nodes is referred to as a community (cluster) in the network. In this work, we use the Louvain method [144] that is implemented in the Gephi [145] tool to detect communities of highly correlated technologies in a TAN. The Louvain method does not require users to specify the number of communities to be detected. It uses an iterative modularity maximization method to partition the network into a finite number of disjoint clusters that will be considered as communities. Each node must be assigned to exactly one community. Intuitively, any edge in a given community has both ends in the same community contributes to increasing modularity, while any edge that cuts across communities has a negative effect on modularity [146].

4.2.3 Determining technology categories

In Stack Overflow, most tags have a brief definition called TagWiki which is collaboratively edited by the community. This mechanism is similar to Wikipedia. According to our observation, the first sentence of the tagWiki always defines the category of this tag. For example, the first sentence of the tag *Matplotlib* is “Matplotlib is a plotting library for Python”⁴. In our recent work [100], we develop the NLP methods to analyze such tag definition sentence to determine the category of a tag. We first carry out Part-of-speech (POS) tagging and phrase chunking to the sentence to get the first noun phrase after the be verb (*is/are*) and then take the last word in the phrase as the category label of the tag. As seen in Figure 5.4, the first phrase after *is* is *plotting library* and the

³The edge is undirected because association rules indicate only the correlations between antecedent and consequent.

⁴<http://stackoverflow.com/tags/matplotlib/info>

FIGURE 4.3: POS tagging and phrase chunking results of the definition sentence of the tag *Matplotlib*

last word in that phrase (i.e., library) is regarded as the category of the tag *Matplotlib*. Interested readers can refer to our paper [100] (Chapter 5) for the technical details and the evaluation of tag category analysis. As there are hundreds of fine-grained categories which will be a distraction for users if we display all categories, we manually categorize them into three general categories: “library”, “language” and “concept”. The library category broadly refers to software library, framework, api, toolkit, wrapper, etc., the language category includes different programming languages, and all others are regarded as concept category such as data structures, algorithms.

4.2.4 Summarizing technology activity

For each technology in a TAN, we summarize the frequency of the corresponding tag used in the set of Stack Overflow questions. We then normalize the frequency over all the technologies in the TAN as a technology activity metric in $(0, 1]$. This technology activity metric is an indicator of the relative community attention to a technology in the TAN, compared with other technologies in the TAN.

4.2.5 Visualizing TAN

We use the Gephi tool [145] to visualize the TAN as follows (see Figure 4.2 for examples). Nodes and edges in one community are shown in the same color⁵. Forceatlas2 layout [147] is used for network spatialization. This layout is especially suitable for inspecting clustering results (i.e., technology communities). The node size represents

⁵The Gephi tool sometimes may assign very similar colors to different communities.

the technology activity metric. That is, the larger the node is, the more questions are tagged with the corresponding technology. The edge length can represent the strength of the corresponding technology associations. Due to the use of Forceatlas2 layout, the edge length bears no meaning in the examples of this chapter.

4.3 The TechLand System

The TechLand system consists of three components: the backend TechLand TAN graph, the frontend browser plugin and the website. In this section, we describe the features of the TechLand system, beginning with the user's interaction with the browser plugin during online search, then detailing the exploration interface of the website.

4.3.1 Augmenting search with information scents

When developers search some technologies on the Web, they often obtain many search results. If the developers are unfamiliar with the technologies, unclear about their goal, or unclear about how to achieve the goal, it can be difficult for them to determine which web pages to read or how to refine the search. Thus, the goal of the TechLand browser plugin is to augment the online search with some information scents.

When a user searches Google, the browser plugin implements a greedy recursive matching method to recognize the technology terms in the query. A technology term can consist of a single word (e.g., javascript, sockets) or a phrase (i.e., data visualization, web scraping). A query can consist of more than one technology term (e.g., javascript data visualization). Given a query, the plugin first splits the query into a list of words by space. Next, it starts with the whole query and recursively searches phrases of consecutive words with shorter length in the technology list of the TechLand graph. The recursion stops when the phrase has a single word. For example, for the query *javascript data visualization*, the plugin recognizes two technology terms, i.e., *javascript* and *data visualization*. As the search is greedy, once the plugin ⁶ recognizes *data visualization*,

⁶https://github.com/tomhanlei/kg_plugin

For the technology terms in the query, the plugin displays the definition of the least frequent technology term (e.g., *data visualization* in *javascript data visualization*), and the TechLand graph for the search terms, in the TechLand panel on the right side of Google search results (see Figure 4.4). The technology description is retrieved from the TagWiki of the corresponding tag on-the-fly using Stack Exchange API⁸. The TechLand graph provides a graphical overview of important technology terms related to the search terms. For example, inspecting the TechLand graph for the query “data visualization”, users can find relevant concepts (e.g., graph-visualization, cluster-analysis), types of data visualization (e.g., bar-chart, tree), data format (e.g., svg, json), and tools for different languages (e.g., JavaScript’s d3.js, highcharts, nvd3.js and dimple.js, C#’s mschart, R’s ggplot2, and Python’s matplotlib). For developers who are unfamiliar with

⁸<https://api.stackexchange.com/>

FIGURE 4.5: The TechLand web interface. 1) The technology definition and more details can be obtained on TagWiki. 2) The monthly asking trend in Stack Overflow. 3) The overall technology associative network and click year to view the TechLand graph for a year. 4) The top-voted questions tagged with *sockets*. 5) A Java code snippet for *sockets* 6) The most-referenced links whose anchor text matches tag name or synonyms.

the data visualization technologies, these information scents can help them select search results to read or refine the search.

4.3.2 Inspecting aggregation of correlated technologies

TechLand visualizes the TechLand graph in an associative network of technologies that are highly correlated with the given technology. The algorithm for mining technology associate network will be explained in the next section. Observing the TechLand graph, users can learn technology description, technologies correlations, and even navigate to the large space of available technologies in an organized way.

Users can interact with the TechLand graph as follows: hover mouse over a technology to highlight its directly correlated nodes; right-click a technology to view its description; double-click a technology to navigate to the TechLand page for that technology.

TechLand clusters technologies in the TechLand Graph as technology clusters [148] and visualizes technology clusters in different colors. Technologies within a cluster are

tightly correlated, while technologies across clusters are not or loosely correlated. For example, Figure 4.4 shows that JavaScript's d3.js is more correlated with graph visualization than other data visualization tools. The node size of a technology is proportional to the technology's co-occurrence frequency with the given technology in Stack Overflow questions. For example, Figure 4.4 shows that when people ask questions about data visualization, more questions are tagged with javascript or d3.js than other technologies. Although by no means conclusive, this indicates that d3.js seems to be more widely adopted for data visualization than other tools.

4.3.3 Comparing technology adoption indicators

TechLand aggregates community question-asking activities on Stack Overflow as factual indicators of technology adoption. This is based on empirical findings which reveal that topics in developers' discuss on Stack Overflow reflect the thoughts, needs and practices of developers [8, 149, 150]. TechLand presents these technology-adoption indicators in three ways for analyzing technology adoption.

First, after constructing the TAN of a given technology, TechLand collects the co-occurrence frequency of a technology in the TAN and the given technology. It then normalizes the co-occurrence frequency over all the technologies in the TAN as a value in $(0, 1]$, and visualizes the normalized value by node size of the technology. For example, comparing node size of d3.js and that of other tools (e.g., mschart, matplotlib, ggplot2), users can see that d3.js co-occurs with data-visualization more frequently in Stack Overflow questions than other tools.

Second, in addition to the overall TAN mined from all questions tagged with a given technology, TechLand also mines the yearly TAN from the questions tagged with a given technology in a particular year. Users can access to these yearly TANs on the TechLand website (Figure 4.5.3). Comparing the evolution of technology landscape for the given technology can reveal the new emerging techniques. For example, from the yearly TANs of data visualization, users can see that d3.js first appears in 2011 and becomes a more and more popular discussion topic on Stack Overflow since then.

FIGURE 4.6: The asking trend comparison among *d3.js*, *matplotlib*, *ggplot2* and *mschart*

Third, TechLand summarizes the number of questions asked per month for a given technology, and produces a monthly asking trend (Figure 4.5.2). Users can compare the asking trend of several technologies on the TechLand website. From the comparison of the asking trend of *d3.js*, *mschart*, *matplotlib*, and *ggplot2* (Figure 4.6), users can spot the fast rising of *d3.js* since 2011, and developers ask more questions about *d3.js* than other tools since 2013.

The above three question-asking activity metrics, although by no means conclusive, can reveal community's interest in certain technologies, and to some extent indicate the amount and quality of documentations for the technologies. They can help users make a choice for the appropriate technologies.

4.3.4 Suggesting relevant web resources

While exploring the technology landscape, users need to frequently search and read some relevant web resources following the information scents collected during exploration. Stack Overflow accumulates a tremendous repository of high-quality web resources for software engineering technologies. These resources include not only the questions and answers themselves, but also other resources (e.g., code examples, hyperlinks) embodied in the questions and answers.

To assist the exploration of relevant web resources for a technology and avoid frequent switching between search engines and the TechLand website, TechLand aggregates the following three types of web resources from Stack Overflow for a given technology:

1) five top-voted questions tagged with the corresponding technology; 2) code snippets from the TagWiki of the corresponding tag; 3) up to five most-referenced hyperlinks whose anchor text matches the tag name or synonyms.

Top-voted questions are retrieved on-the-fly using Stack Exchange API when users interact with the TechLand website. For example, in the top-voted questions for sockets (Figure 4.5.4), users can find Stack Overflow discussion threads on “What is the difference between a port and a socket?”, and “Can two applications listen to the same port?”. In the code snippet for sockets (Figure 4.5.5), users can find a code example that demonstrates sockets programming in Java. In the popular links (Figure 4.5.6), users can find web resources to implement sockets in different programming languages such as python, c#, java, actionscript and Android platform. Details of extracting most-referenced hyperlinks and code snippets will be explained in the next section.

In fact, such community-highly-recognized web resources are often in the top search results by Google and users can find similar things using search engines. However, direct access to these web resources within the TechLand website provides an alternative to search while exploring the technology landscape on the TechLand website at the same time.

4.3.5 The usage of TechLand website

We develop a TechLand website⁹ that displays a technology page for a given technology in the mined technology landscape. The technology page shows the technology description extracted from the TagWiki, the mined TAN, and other related information extracted from the Stack Overflow. The user can search the technologies in the mined technology landscape or navigate from one technology page to another in the graphical TAN. The website also allows the user to compare the TAN of several technologies side by side.

We release our website to the public and post this news on several programming-related websites (e.g., <http://stackapps.com/questions/6569>). According to the

⁹<https://graphofknowledge.appspot.com/>

FIGURE 4.7: The Google Analytics of our website

Google Analytics¹⁰, 3,528 users from 108 countries visit our site (Figure 4.7) from Sept 4th 2015 to Oct 4th 2017. These users on average browse 3.08 pages in each session for 3.5 minutes and they browse 15,580 pages in total (including the homepage). The usage statistics show initial evidence of the public interests in technology landscape services.

4.4 Empirical Study

We conduct empirical evaluation of our approach and the mined TANs using Stack Overflow data dump. In particular, we investigate the following research questions:

- RQ1: How do different mining thresholds affect the size and modularity of the mined TAN?
- RQ2: Can the mined TAN capture the important technologies from a majority of Stack Overflow questions?
- RQ3: Are the mined technology associations semantically related?

¹⁰As most search engine robots do not activate Javascript, robot traffic is not counted in Google Analytics [151]

4.4.1 Dataset

In this study, we use the Stack Overflow data dump released in March 2015. The data ranges from 2008-07-31 to 2015-03-08 and contains 7.89 million questions that are attached with 2 or more tags, and 39948 unique tags from these questions. These questions and tags constitute the dataset for our evaluation.

4.4.2 RQ1: Size and modularity of technology communities

Given a minimal support, the number of technology associations in the mined TAN is affected by the minimal confidence t_{conf} . The number of technology associations consequently affects the size and modularity of technology communities in the mined TAN. Next, we analyze the impact of the minimal confidence t_{conf} on the size and modularity of the general TAN mined at the minimal support 0.0007 and the 11 different minimal confidences t_{conf} (0 to 0.5 with increment 0.05).

As shown in Figure 4.9(a) and Figure 4.9(b), the number of edges (i.e., technology associations) keeps decreasing, as the minimal confidence increases. In contrast, the number of nodes (i.e., technologies) remains unchanged until the minimal confidence increases to certain extent (0.2 in this case). After that, the number of nodes decreases roughly at the same pace as the number of edges decreases. This suggests that the increase of minimal confidence has more impact on the structure of the mined TAN than the nodes of the TAN.

We compute the density of the mined TAN (i.e., the number of edges in the TAN divided by the number of edges in a complete graph of the same number of nodes), and the average of local node connectivity (i.e., minimum number of nodes that must be removed to disconnect two nodes) of all pairs of nodes in the TAN. As shown in Figure 4.9(c) and Figure 4.9(d), the density of the knowledge graph remains low and relatively stable as the minimal confidence increases. However, the node connectivity drops sharply as the minimal confidence increases.

As shown in Figure 4.9(e) and Figure 4.9(f), the decrease of node connectivity in turn results in the increase of the number of technology communities and the modularity of technology communities in the TAN. Figure 4.9(g) shows the box plot of the number of tags in the detected technology communities at different minimal confidences. We can observe a trade-off between the size and modularity of technology communities. At low minimal confidence, the knowledge graph has more weak technology associations, which often results in small numbers of large communities with low modularity.

The increase of the minimal confidence can remove the weaker associations from the TAN with higher modularity. As a result, the knowledge graph becomes less connective. However, a very high confidence risks throwing away meaningful technology associations, leading to excessive partition of the TAN into many small, disconnected communities, which is often not desirable. Therefore, to produce a good balance and trade-off between the number of edges and nodes in the general TAN, the minimal confidence should be between 0.15 and 0.25.

4.4.3 RQ2: Coverage of tags and questions

The number of technologies in the mined TAN is affected by the minimal support t_{sup} and the minimal confidence t_{conf} . When t_{conf} is set to 0, all the frequent pairs of tags at a given minimal support will be included in the TAN, and thus the TAN will have the maximum number of technology at a given minimal support. This TAN defines the upper bound of the coverage of tags and questions at a given minimal support t_{sup} , which will be evaluated in this section. In particular, we evaluate the general TAN mined at the 10 minimal support t_{sup} (0.0001 to 0.001 with increment 0.0001)¹¹. For the following sections, we will use the general TAN mined at the minimal support 0.0007, because the resulting general TAN is complex enough to analyze the key characteristics of the mined TAN, meanwhile it can be clearly visualized in the thesis (see Figure 4.2).

¹¹Due to the data sparsity, too large support value results in rather small and sparse TAN. After experiments with various values, we choose this range for our evaluation.

FIGURE 4.8: The coverage of the general TAN

4.4.3.1 Coverage of questions

If the N tags of a question appear in the TAN, we say that the question is covered by N technologies in the TAN. Figure 4.8(b) shows the percentage of the questions in our dataset that are covered by 1-5 technologies at the 10 minimal supports. Note that the percentage is computed in an exclusive manner. That is, the questions that are covered by N technologies do not include those that are covered by $N - 1$ technologies.

We can see that although the general TAN covers only a small portion of all the tags in our dataset, it still covers a large portion of all the questions. As the minimal support increases, the coverage of questions by 3 or more technologies decreases significantly from 40% at 0.0001 to 10% at 0.001. The coverage of questions by 2 tags remains about 30% at different minimal supports. The coverage of questions by only 1 tag increases from 22% at 0.0001 to 41% at 0.001. The overall coverage of questions decreases from 96% at 0.0001 to 82% at 0.001. This suggests that most of the questions that can be covered by 3 or more technologies at lower minimal support can still be covered by the TAN at higher minimal support, but at high minimal support these questions can only be covered by 1 or 2 most frequently used tags that are used to tag large numbers of questions.

4.4.3.2 Coverage of tags

To examine the coverage of tags, we rank all the tags in our dataset by their usage frequency in the set of Stack Overflow questions. We scan the ranked list of all the tags to find the tags that appear in the mined TAN at a given minimal support. We truncate the ranked list at the Lowest Rank Position (LRP) of the technologies in the

FIGURE 4.9: The impact of minimal confidence on the size and modularity of the general TAN

TAN. Figure 4.8(a) presents the analysis results at the 10 minimal supports. Tags are ranked from left to right by decreasing frequency of use. A red line indicates that the tag ranked at this position is in the TAN at a given minimal support, while a pink line indicates that the tag is not in the TAN.

Figure 4.8(a) shows that the number of tags (#T) in the TAN and the LRP of these tags differ greatly at different minimal supports. Note that we scale the visualization of tag coverage at different minimal support to facilitate the observation of tag coverage patterns at different minimal supports. Overall, the mined TAN captures a meaningful conceptualization of important technologies than the individual tags alone. We use the general TAN mined at the minimal support 0.0001 (i.e., the bar at the bottom in Figure 4.8(a)) as an example for detailed discussion.

The general TAN mined at the minimal support 0.0001 contains 1,439 tags (#T). These 1,439 tags account for 53% of the top 2,714 most frequently used tags (LRP). 66% of these 1,439 tags fall into the top 40% range of the 2,714 most frequently used tags. However, about 13% of the top 40% of the 2,714 most frequently used tags do not appear in the general TAN, as indicated by the pink lines within the top 40%. These tags are usually some common programming concepts such as *formatting*, *automation*, *documentation* and *numbers*. Although these common tags are frequently used as a whole, their co-occurrences with other tags are often not frequent enough because they are correlated with many technologies. As such, these common tags do not appear in the general TAN. Note that these common tags may still appear in the technology-specific TAN if their co-occurrences with a given technology is frequent enough.

34% of the 1439 tags in the general TAN scatter in the lower 60% range of the 2,714 most frequently used tags. These tags usually represent features of some specific techniques, such as *django-queryset*, *android-custom-view* and *jquery-ui-draggable*. Although these tags are less frequently used than many other frequently used tags, their co-occurrences with some specific technologies (e.g., *django*, *android*, *jquery*) are often frequent. As such, these tags appear in the TAN.

4.4.4 RQ3: Semantic distance of technology associations

In this section, we examine whether technology associations are meaningful by evaluating the semantic distance between the two correlated technologies in the mined TAN using the Google Trends [152]. Google Trends¹² is a public web facility of Google Inc., based on Google Search, that shows how often a particular search-term is entered. The assumption is that the co-occurrence of a set of words in the same queries is a good indicator of the relatedness between the words.

Given a technology association (i.e., an edge $\langle t_1, t_2 \rangle$) in the TAN, we generate a set of search terms to query Google Trends. For example, to check whether the two technologies *php* and *facebook* are really correlated, we query the Google Trends with the search terms “php facebook”. Google Trends provide the trend statistics for popular queries. For example, “php facebook” is a popular query because Facebook is built using PHP and it supports PHP APIs. If a set of search terms is not popular enough, Google Trends will provide no trend statistics.

As shown in Figure 4.9(a), there are a small percentage of technology associations (less than 10% at all the minimal confidences) in the TAN, which are not present in Google Trends. Lower minimal confidence values do not significantly result in more noisy technology associations. Furthermore, even the technology associations are not present in Google Trends, it does not necessarily indicate wrong associations. Take the minimal confidence 0.15 as an example. The TAN has 15 technology associations that are not present in Google Trends. 7 out of these 15 associations involves tags with specific

¹²https://en.wikipedia.org/wiki/Google_Trends

version number such as $\langle doctrine2, symfony2 \rangle$ which is not commonly searched in Google. In contrast, “doctrine symfony2” is a popular query. The other 8 associations are the results of different wording styles in Google and Stack Overflow. For example, Stack Overflow users frequently tag questions with both *knockout.js* and *javascript*, while Google users search “knockoutjs” directly without “javascript”.

4.5 User Study

The user study is designed as a comparative study to measure the quality of information collected by users. We compare Google search with the TechLand system support (i.e., experimental system (E)) against using Google search only (i.e., baseline system (B)). In experiments the participants are provided with a scenario describing information needs and asked to use the system to acquire information.

4.5.1 Tasks

To demonstrate the ability of TechLand to support diverse tasks, we simulate three types of technology-landscape exploration tasks: practice-driven, concept-driven, and tool-driven. Each type has two tasks and each task has three questions. The tasks and their questions are shown in Table 4.1. The first two questions of all the tasks are generic, which ask the participants to find related concepts, languages, and tools for a given scenario. The third question of the tasks are specific to the technology in a given scenario, which requires deeper understanding of available technologies, their correlations, and their adoption. These questions represent the range of developers’ information needs in technology landscape exploration. Furthermore, the questions are designed to motivate the participant to use all the features of the TechLand system.

Task 1: bdd (behavior driven development)
(1) List 5 related concepts; (2) Find 5 PLs and their frameworks/library to support BDD practice; (3) From the languages and tools in question (2), please determine BDD is more widely adopted for applications developed in which programming languages? Rank those programming languages in order.
Task 2: data visualization
(1) List 5 related concepts; (2) Determine 4 PLs and corresponding tools/library for data visualization; (3) Assume that you know all these languages, which language and tool in Question (2) will you use? Please explain your reasons such as documentation, community adoption.
Task 3: encryption
(1) List 5 related concepts; (2) Find 5 PLs and their frameworks/library to implement encryption methods; (3) Assume that you know both java and c#, is there any tool that can be used for both languages?
Task 4: statistics
(1) List 5 related concepts; (2) Select 2-3 PLs and corresponding tools/library that support statistical analysis; (3) Assume that you know all these languages, which language and tool in Question (2) will you use? Please explain your reasons such as documentation, community adoption.
Task 5: beautifulsoup
(1) List 5 related concepts; (2) List 2-3 tasks that beautifulsoup can support; (3) For tasks that you find in question (2), are there any alternative tools to beautifulsoup? (not limited to the same programming language)
Task 6: numpy
(1) List 5 related concepts; (2) List at least 3 libraries that are highly related with numPy; (3) What tasks can these libraries in question (2) implement? (please list them one by one)

TABLE 4.1: Task Descriptions (PL denotes programming language)

4.5.2 Participants

We recruit 12 participants by promoting the experiments via emails and word-of-mouth. The recruited participants are all PhD students in our school majoring in computer science and computer engineering. The participants have diverse research background and they use different software tools and programming languages in their research work. All the participants use Google regularly. However, none of the participants claim to be familiar with the technologies in the experimental tasks. The reason for selecting this set of participants is that in this formative evaluation we would like to focus on the ability of the TechLand system to support users who are unfamiliar with a technology.

We would like to elicit the information needs of these users in technology landscape exploration, and investigate to what extent the TechLand system can satisfy their needs, and how their needs can be better supported. The participants receive \$10 shopping coupon as a compensation of their time.

4.5.3 Experiment procedures

The experiment begins with an introduction to the study. Then, the participants are given 10-minute tutorial about the TechLand system features, and perform a training task for 20 minutes to become familiar with the system. After the training session, each participant is asked to work on the six tasks. All of the six tasks are completed with no interventions by the experimenters. For each type of the tasks, the participants use the experimental system for one task and the baseline system for the other. The order of task types, the order of tasks for each type, and the order of using the experimental or baseline system are rotated based on the Latin Square, which help reduce the learning and fatigue effects.

The time allocated for each task is 10 minutes. At the end of each task, the participants are asked to rate their confidence in the information they collect (on 5-point likert scale with 1 being least confident and 5 being most confident). If the task uses the experimental system, they are also be asked to rate the helpfulness of the TechLand system for the task (5-point likert scale with 1 being least helpful and 5 being most helpful). Once they complete all six tasks, the participants fill in the System Usability Scale (SUS) questionnaire [153]. The questionnaire also asks the participants to select the TechLand system features that they deem most useful or least useful for the tasks. In addition, we conduct post-test interviews to obtain users qualitative opinions about the interaction techniques and experience of the TechLand system.

4.5.4 Results

We evaluate the TechLand system in terms of both system performance and system usability. For system performance, we compare results confidence, results completeness

Measures	Google	Google + TechLand
Average confidence	3.56(0.93)	4.0(0.75)*
Average completeness	0.922(0.113)	0.988(0.056)**
Average validity	0.860(0.139)	0.958(0.074)**

TABLE 4.2: The mean (standard deviation) of the three measures, * denotes $p < 0.1$ and ** denotes $p < 0.05$.

and results validity. We use the Wilcoxon Signed-Rank test to check the significance of the differences between the experimental system and the baseline system. For system usability, we present the average value for each usability question.

4.5.4.1 System performance

For the first and second question of each task, we compute the question completeness score as the number of entries the participants collect for the question divided by the number of entries requested by the question. For example, if the question asks for 5 related concepts and the participant finds 3 concepts, the result completeness for this question is 0.6. For the third question of each task, if the participant answers the question, we set the question completeness score as 1, otherwise 0. We then obtain the results-completeness score for the task by averaging the three question-completeness scores. We find a domain expert in our school for each task to examine the results validity. The rating procedure is similar to results completeness, but the expert needs to examine whether the collected information is correct and assign a score in $[0 - 1]$.

Table 4.2 shows the average user confidence in their results, and the average results completeness, and the average results validity. The *average confidence* shows that users' confidence is higher when using Google and our TechLand system. This weakly demonstrate the effectiveness of our system (due to significance value). Furthermore, the average helpfulness of the TechLand system for the tasks is 4.09. This also demonstrates the effectiveness of the TechLand system. There are significant difference in *average completeness* and *average validity* between using Google+TechLand and using Google only. On average, the results completeness and results validity of using

FIGURE 4.10: Average score of SUS results

Google+TechLend increase 7.2% and 11.4% respectively, compared with those of using Google only.

4.5.4.2 System usability

Figure. 4.10 summarizes the users' ratings of system usability. Figure 4.10(a) shows that users agree or strongly agree that our system is easy to use and the features of the Techland system are well integrated. Figure. 4.10(b) further confirms the simplicity and consistency of our Techland system.

4.5.4.3 User feedback

In post-study questionnaires, the participants rate the TechLand graph most useful for the tasks, followed by top-voted questions. They suggest that three features are needed to make the TechLand graph more useful: technology categorization, explaining why two technologies are correlated, and making analogical relations between technologies explicit.

The participants rate popular links and code snippets least useful and rarely use them. Popular links overlaps Google search results, and thus offer little new value. The nature of the experimental tasks is about high-level background and comparison of some technologies. In such tasks, users do not need fine-grained information such as code snippets.

The participants suggest that asking trend of an individual technology is not useful, but comparing the trends of several technologies is a good indicator of technology adoption.

However, they feel inconvenient to switch to another webpage to see trend comparison. They wish that the system could automatically provide trend comparison while they search or investigate the technologies.

4.5.5 Limitations

Our evaluation has several limitations. The participants are PhD students. The study is done in a controlled experiment. First-use studies make it difficult to understand how a user might use the system longitudinally. The experimental results may not be generalizable to other populations or longitudinal use in the field. We release TechLand publicly to collect real-world usage data to answer these questions.

4.6 Conclusion

In this chapter, we present a data mining technique for mining TAN which can be regarded as a naive knowledge graph from the “by-product” (i.e., tags) of the Q&A practices in community Q&A websites. Our evaluation using Stack Overflow data dump shows that the mined technology landscape can provide an aggregated view of a wide range of technologies, the complex relationships among the technologies, and the trend of the technologies, which reflect the practices of a large community of developers. We also introduce our website for accessing the mined technology landscape. The website usage data, albeit limited, provides initial evidence of the interests in and the usefulness of the mined technology landscape.

Although the current knowledge graph can provide direct answers for some high-level questions such as technologies for certain tasks or relationships among some technologies, it is still too limited to answer more complicated questions. So the further enrichment of the basic knowledge graph is needed.

Chapter 5

Mining Analogical Libraries from Q&A Discussions

5.1 Introduction

Third-party libraries are an integral part of many software systems. Developers do not need to reimplement the wheels by using libraries which provide robust and efficient functionalities. Thung et al. [71] show that among 1,008 projects in GitHub they investigate, 93.3% of which use third-party libraries, at an average of 28 third-party libraries per project. As a software project and the libraries used in the project inevitably evolve in their own direction, developers sometimes need to replace some currently-used libraries by another ones. For example, the library that a developer currently uses is no longer under active development, or lacks certain desired features, or cannot satisfy performance requirements. In such cases, the developer wants to find some good replacements [72]. In other cases, the developer switches to a new programming language, but he would like to “reuse” his good experience with some libraries that he is familiar with [69, 70] (such as the example shown in Fig. 5.1). Even though some libraries provide interfaces to multiple programming languages, most libraries are implemented in only one language and work the best with that language. It would be desirable to

FIGURE 5.1: An example of analogical-library question in Stack Overflow.

find analogical libraries that are best suited for the new programming language that the developer switches to.

In this chapter, we are trying to mine the analogical relationships among third-party libraries which are already inside our knowledge graph built in last chapter. By enriching the basic knowledge graph with this fine-grained relationships, our knowledge graph can now answer queries about recommending similar third-party libraries with direct answers.

Developers can search the Web for analogical libraries that can provide features comparable to the libraries they are already familiar with. They could find useful information in some community-curated list of libraries, such as *unit testing framework on Wikipedia*, and *Awesome PHP on Github*. These library lists are usually very comprehensive, but they often contain many isolated libraries that may distract developers. Developers may also find useful information in blogs (e.g., “*Beyond JUnit – Testing Frameworks alternatives*”) or forum posts (e.g., “*Alternatives to JUnit*”). Blogs and forum posts are usually more focused, but they are often opinion-based and many past posts contain out-of-date information. When developers cannot find satisfactory information on the Internet or want to confirm their search findings, they may ask on Q&A web sites like Stack Overflow (e.g., Fig. 5.1), but may not get the immediate answers.

Although many research works have been carried out for mining similar code snippets [67], functions [68], or APIs [69, 70], finding analogical libraries for different programming languages or mobile platforms are rarely investigated. A key challenge in analogical-library recommendation is that the program analysis (based on code) or

FIGURE 5.2: The similar context of Python’s nltk and Java’s opennlp in tag sentences

information retrieval (based on text) methods cannot properly model the semantics of libraries for reasoning their analogical relations.

In this chapter, we present a new approach to find analogical libraries and enrich the existing knowledge graph mentioned in Chapter 4 with the analogical relationships, as these libraries are already in the knowledge graph. Our approach is based on the empirical findings showing that taken in aggregate posts on Stack Overflow act as a knowledge repository of developers’ practices and thoughts [8], and that the main technologies or constructs that a question revolves around can usually be identified from question tags [54] (see Fig. 5.1). Instead of listing dozens of abandoned libraries or relying on blogs or Q&A posts, our approach recommends analogical libraries based on a knowledge base of analogical libraries mined from tags of millions of Stack Overflow questions. This knowledge base can be periodically updated with new Stack Overflow questions, and thus is like forever evolving “blog posts” about good analogical libraries to the libraries that one is familiar with.

Our approach is motivated by the recent success of neural network language models in Natural Language Processing (NLP) applications [74, 154]. Recently, Mikolov et al. [155] and Turney [75] demonstrate that neural network language models are able to learn word representations (or word embeddings) that can be exploited to solve analogy questions of the form “a is to A as ? is to B”, for example, “Paris is to France as ? is to Spain”. The unknown word “?” can be inferred from the words (e.g., Madrid) whose word embedding is most similar to the resulting vector of vector arithmetic $a - A + B$ (e.g., $Paris - France + Spain$)

In our approach, we consider tags of a Stack Overflow question as a tag sentence, and each tag as a word in the sentence. As illustrated in Fig. 5.2, analogical libraries (such as Python’s *nltk* and Java’s *opennlp*) would share similar context in tag sentences, such as common concepts and techniques. Given a corpus of tag sentences derived from Stack Overflow questions, we use continuous skip-gram learning algorithm [74] to learn the tag embedding of each tag using the surrounding context of the tag in the corpus of tag sentences. Given a library (e.g., *python*’s *nltk*), we reduce the problem of finding analogical libraries for a programming language (e.g., *java*) as *nltk* for *python* to a K-nearest-neighbor search for the tags (e.g., *opennlp*) whose word representation is the most similar to the vector $nltk - python + java$ in the resulting word embedding space.

However, directly applying neural network language models to the problem of learning tag embeddings in tag sentences, as opposed to learning word representations in everyday NLP problems, brings unique challenges. First, contrary to everyday language where linguistic rules and notions of words and sentences are clearly defined, question tags on Stack Overflow are composed of only up-to five terms where there is no existing notion of the surrounding context equivalent to natural language domain. Second, question tags could be noisy or biased such that they cannot reflect the inherent relationship between tags and further mislead the learning process.

To address these challenges, we incorporate domain-specific relational and categorical knowledge into tag embeddings in order to produce better mappings of analogical libraries. In our approach, relational knowledge encodes the correlation between tags. We use association rule mining [143] to mine the correlation between tags from tag co-occurrence patterns in millions of Stack Overflow questions. Categorical knowledge encodes the category of tags (e.g., library, framework, concept, platform, database, and so on). We use Part-of-Speech tagging and phrase chunking methods [156] to analyze the TagWiki description of each tag to determine the category of the tags. Both relational and categorical knowledge can serve as valuable external information to help differentiate library-language/platform pairs with analogy relationships, even if there is little context information or biased/noisy context information in tag sentences.

Apart from building a knowledge base of analogical libraries across different programming languages, we also extend our method for analogical-libraries recommendation for different mobile platforms such as iOS, Android and Windows-Phone. Furthermore, after obtaining a list of recommended libraries, developers usually need to determine which one is the most suitable one for their work. To that end, they need to refer to some existing articles about the comparison between a library and a recommended analogical library. To satisfy this post-recommendation information need, we develop a keyword-matching method to extract such comparison questions and answer snippets in Stack Overflow discussions.

We implement our approach in a proof-of-concept web application¹ for programming-language based recommendation and for mobile-platform based recommendation². The application takes as input a library name and recommends analogical libraries for different programming languages or different platforms. We evaluate the analogical-libraries recommendations for randomly selected 140 libraries using our approach. The results show that our approach can make accurate recommendation of analogical libraries for both different programming languages and mobile platforms. Furthermore, Google Analytics of our website traffic provides initial evidence of the potential usefulness of our web application for software developers. Before August 2017, more than 34.7 thousand users from 168 countries have visited our site for analogical libraries.

Our contributions of this work is summarized as follows:

- Incorporating the semantical, relational and categorical information, we build a knowledge of analogical libraries across different programming languages and mobile platforms to enrich the existing knowledge graph.
- To make our approach a more complete solution for finding analogical libraries, we develop a keyword-matching method to extract comparison questions and answer snippets in Stack Overflow that help developers compare recommended libraries.

¹<https://graphofknowledge.appspot.com/similartech>

²<https://graphofknowledge.appspot.com/similarmobiletech>

FIGURE 5.3: The overview of our approach

- We evaluate the usefulness of our analogical-libraries recommendations by comparing our recommendations with user-provided answers to 70 analogical-libraries-related questions in Stack Overflow.
- Our web application receives steady visits (on average 1.1K per month) since its launch in November 2015. We log the user visit behavior and analyze the users' logs for insights.

5.2 The Approach

Our approach takes as input the tags of each question in Stack Overflow and the TagWiki of each tag, and produces as output a knowledge base of analogical libraries (Fig. 5.3). Our approach considers the tags of a Stack Overflow question as a tag sentence, and each tag of the question as a word in the tag sentence. Given a set of Stack Overflow questions, we build a corpus of tag sentences, one tag sentence per question. Given the corpus of tag sentences, we use association rule mining [143] to mine the correlation between tags (Section 5.2.2), and use continuous skip-gram model [74] to learn tag embeddings (Section 5.2.3). We develop POS tagging and phrase chunking methods to analyze the tag definition in the TagWiki of each tag to determine the tag category (Section 5.2.1). Tag embeddings and relational and categorical knowledge of tags are

FIGURE 5.4: POS tagging and phrase chunking results of the definition sentence of the tag *iOS*

incorporated to build the knowledge base of analogical libraries for different programming languages or mobile platforms (Section 5.2.4). Given a query library, our approach returns analogical libraries based on this knowledge base. Furthermore, to help developers understand the recommended libraries, our approach formulates queries for a given pair of analogical libraries and search Stack Overflow posts for questions and answer snippets that likely compare the two libraries (Section 5.2.5).

5.2.1 Mining categorical information

In Figure 4.2, we can see that the tags can be of different categories, such as programming language, library, framework, tool, IDE, operating systems, etc. To determine the category of a tag, we resort to the tag definition in the TagWiki of the tag. The TagWiki of a tag is collaboratively edited by the Stack Overflow community. Although there are no strict formatting rules in Stack Overflow, the TagWiki description usually starts with a short sentence to define the tag. For example, the tagWiki of the tag *iOS* starts with the sentence “iOS is a mobile operating system developed by Apple”. Typically, the first noun phrase just after the *be* verb defines the category of the tag. For example, from the tag definition of *iOS*, we can learn that the category of *iOS* is *operating system*.

Based on this heuristic, we use the NLP methods (similar to the methods used in [156] for named entity recognition) to extract such noun phrase from the tag definition sentence as the category of a tag. Given the tagWiki of a tag in Stack Overflow, we extract the first sentence of the TagWiki description, and clean up the sentence by removing hyperlinks and brackets such as “{ }”, “()”. Then, we apply Part of Speech (POS) tagging

and phrase chunking to the extracted sentence. POS tagging is the process of marking up a word in a text as corresponding to a particular part of speech, such as noun, verb, adjective. Phrase chunking is the process of segmenting a sentence into its sub-constituents, such as noun phrases, verb phrases. We use the Python NLTK library³ for POS tagging⁴ and phrase chunking [157]. Fig. 5.4 shows the results for the tag definition sentence of *iOS*. Based on the POS tagging and phrase chunking results, we extract the first noun phrase (NP) (*operating system* in this example) after the be verb (*is* in this example). We use this noun phrase as the category of the tag. That is, the category of *iOS* is *operating system*.

With this method, we obtain 318 categories for the 19,573 tags (about 54% of all the tags that have TagWiki). We manually normalize these 318 categories labels, such as merging *operating system* and *os* as *os*, normalizing uppercase and lowercase (e.g., *API* and *api*). As a result, we obtain 167 categories. Furthermore, we manually categorize these 167 categories into four general categories: programming language, platform, library, and concept/standard. These four general categories are defined in our previous work for named entity recognition [158]. This generalization step is necessary, especially for the library tags that broadly refer to the tags whose fine-grained categories can be library, framework, api, toolkit, wrapper, and so on⁵. This is because the meaning of these fine-grained categories is often overlapping, and there is no consistent rule for the usage of these terms in the TagWiki. For example, in Stack Overflow’s TagWiki, *junit* is defined as a framework, *google-visualization* is defined as an API, and *wxpython* is defined as a wrapper. All these tags are referred to as library tags in our approach.

Although the above method obtains the tag category for the majority of the tags, the first sentence of the TagWiki of many tags is not formatted as “tag be noun phrase” form. For example, the first sentence of the TagWiki of the tag *itext* is “Library to create and manipulate PDF documents in Java”, or for *markermanager* is “A Google Maps tool”, or for *ghc-pkg* is “The command ghc-pkg can be used to handle GHC packages”. As

³http://www.nltk.org/_modules/nltk/tag.html

⁴https://www.ling.upenn.edu/courses/Fall_2003/ling001/penn_treebank_pos.html

⁵A complete list can be found at <https://graphofknowledge.appspot.com/libCategory>

there is no *be* verb in this sentence, the above NLP method cannot return a noun phrase for the tag category. According to our observation, for most of such cases, the category of the tag also appears in the sentence, but in many different ways. And it is very likely that the category word appears at the start of the definition sentence. Therefore, we use a dictionary look-up method to determine the category of such tags. Specially, we use the 167 categories obtained using the above NLP method as a dictionary to recognize the category of the tags that have not been categorized using the NLP method. Given an uncategorized tag, we scan the first sentence of the tag's TagWiki from the beginning, and search for the first match of a category label in the sentence. If a match is found, the tag is categorized as the matched category. For example, the tag *itext* is categorized as *library* using this dictionary look-up method. Using the dictionary look-up method, we obtain the category for 11,059 more tags.

Note that we cannot categorize some (less than 15%) of the tags using the above NLP method and the dictionary look-up method. This is because these tags do not have a clear tag definition sentence, for example, the TagWiki of the tag *richtextbox* states that “The RichTextBox control enables you to display or edit RTF content”. This sentence is not a clear definition of what *richtextbox* is. Or no category match can be found in the tag definition sentence of some tags. For example, the TagWiki of the tag *carousel* states that “A rotating display of content that can house a variety of content”. Unfortunately, we do not have the category “display” in the 167 categories we collect using the NLP method. When building analogical-libraries knowledge base, we exclude these uncategorized tags as potential candidates.

5.2.2 Mining relational information

In Stack Overflow, each question has up to 5 tags. These tags usually identify the main technologies and constructs that the question revolves around [54] (see Fig. 5.1 for an example). As Stack Overflow manages question tags as a set of terms, the correlation between tags are implicit. We use association rule mining [143] to discover important correlation between tags.

In our application of association rule mining, we regard each tag sentence as a transaction, and each tag in the sentence as an item in the transaction. There are two parameters in association rule mining:

$$\text{support}(t_i, t_j) = \frac{\#tagSent \text{ containing } (t_i \text{ and } t_j)}{\#tagSent}$$

$$\text{confidence}(t_i \Rightarrow t_j) = \frac{\#tagSent \text{ containing } (t_i \text{ and } t_j)}{\#tagSent \text{ containing } t_i}$$

where t_i and t_j are two different tags, and $tagSent$ is a tag sentence. The *support* value measures how frequent the two tags co-occur in all the tag sentences. The *confidence* value measures the proportion of the tag sentences containing both t_i and t_j compared with all the tag sentences containing t_i .

If the support value and confidence value of a tag pair $\{t_1, t_2\}$ are above the respective threshold t_{sup} and t_{conf} , we obtain an association rule $t_1 \Rightarrow t_2$. Given the mined association rules between tags, we construct a tag correlation graph. The tag correlation graph is an undirected graph $G(V, E)$, where the node set V contains the tags appearing in the association rules, and the edge set E contains edges $\langle t_1, t_2 \rangle$ if the two tags has the association rule $t_1 \Rightarrow t_2$, $t_2 \Rightarrow t_1$ or both. Within the knowledge graph, according to the edges, we can know that if one certain library is related to the programming language or not.

Note that we do not use frequent itemset mining because the confidence value in association rule is very important to remove noisy relations. For example, although *nltk* is a library of *python*, it also co-occurs with *java* frequently because many questions asks is it possible to use it in *java* environment or counterpart libraries. Although their support value is larger than the threshold, their confidence value (i.e., compared with other tags co-occur with *java*), it is much lower than the threshold, leading to the ignorance.

FIGURE 5.5: The continuous skip-gram model. It predicts the surrounding words given the center word.

5.2.3 Learning tag embeddings

Word embeddings are low-dimensional vector representations of words that are built on the assumption that words with similar meanings tend to present in similar contexts. In our approach, given a corpus of tag sentences, we use continuous skip-gram learning algorithm [74] to learn the word representation of each tag using the surrounding context of the tag in the corpus of tag sentences. In the resulting word embedding space, the vector offsets between analogical libraries and their corresponding programming languages or mobile platforms would exhibit *relational similarity*, for example, $nltk - python \approx opennlp - java$, $afnetworking - ios \approx volley - android$. Thus, given a library (e.g., *python*'s *nltk*), we can infer analogical libraries for a programming language (e.g., *java*) as *nltk* for *python* by a K-nearest-neighbor search for the tags (e.g., *opennlp*) whose word representation is the most similar to the vector $nltk - python + java$ in the resulting word embedding space. In the same way, we can infer analogical libraries for different mobile platforms.

Continuous skip-gram model [74] is a recently proposed algorithm for learning word embeddings using a neural network model. As illustrated in Fig. 6.3, the objective of the continuous skip-gram model is to learn the word representation of each word that is good at predicting the co-occurring words in the same sentence. Specifically, given a sequence of training text stream $t_1, t_2, \dots, t_k$, the objective of the continuous skip-gram model is to maximize the following average log probability:

$$L = \frac{1}{K} \sum_{k=1}^K \sum_{-N \leq j \leq N, j \neq 0} \log p(t_{k+j} | t_k) \quad (5.1)$$

where t_k is the central word, t_{k+j} is its surrounding word with the distance j , and N indicates the context window size to be $2N + 1$. In our application of the continuous skip-gram model, a tag sentence is a training text stream, and each tag is a word. As tag sentence is short (has at most 5 tags), we set N at 5 in our approach so that the context of one tag is all other tags in the current sentences. That is, the context window contains all other tags as the surrounding words for a given tag.

The probability $p(t_{k+j} | t_k)$ in Eq. 6.3 can be formulated as a log-linear softmax function which can be efficiently solved by the negative sampling method [74].

$$p(t_{k+j} | t_k) = \frac{\exp((v'_{t_{k+j}})^T v_{t_k})}{\sum_{t=1}^V \exp((v'_t)^T v_{t_k})} \quad (5.2)$$

where v_t and v'_t are the input and the output latent variables, i.e., the input and output representation vectors of t , and V is the vocabulary size. However, the computation cost of the formulation 5.2 is proportional to the vocabulary size V which is often very large. To calculate it efficiently, negative sampling [74] is adopted so that the training time yields linear scale to the number of noise samples and it becomes independent of the vocabulary size. After the iterative feed-forward and back propagation, the training process finally converges, and each tag obtains a low-dimension vector as its word representation (i.e., tag embedding) in the resulting vector space.

5.2.4 Building analogical-libraries knowledge base

We build a knowledge base of analogical libraries for different programming languages (or mobile platforms) by incorporating tag embeddings and categorical and relational knowledge of tags. First, we obtain the initial analogical-library candidates for different programming languages (or mobile platforms) based on tag embeddings and tag categories. Then, we refine the initial candidates according to the correlations of libraries and programming languages (or mobile platforms).

5.2.4.1 Obtain analogical-library candidates

Given a library tag t_1 , we first examine its correlated tags to determine its base programming language (or mobile platform), denoted as $base_1$. Let $base_2$ be a programming-language (or mobile-platform) tag which can be the same as $base_1$ or be different from $base_1$. Let $vec(x)$ be the tag embedding of the tag x . To find the analogical libraries t_2 for the $base_2$ as the library t_1 for the $base_1$, we find the library tags t_2 whose tag embedding $vec(t_2)$ is most similar (by cosine similarity in this work) to the vector $vec(t_1) - vec(base_1) + vec(base_2)$, i.e.,

$$\operatorname{argmax}_{t_2 \in T} \cos(vec(t_2), vec(t_1) - vec(base_1) + vec(base_2)) \quad (5.3)$$

where T is the set of library tags excluding t_1 , and $\cos(u, v)$ is the cosine similarity of the two vectors.

Note that tags whose tag embedding is similar to the vector $vec(t_1) - vec(base_1) + vec(base_2)$ may not always be library tags. For example, tag embeddings of the tags *nlp*, *named-entity-recognition* and *language-model* are similar to the vector $vec(nltk) - vec(python) + vec(java)$. These tags are relevant to the *nltk* library as they refer to some NLP concepts and tasks, but they are not analogical libraries to the *nltk*. In our approach, we rely on the category of tags (i.e., categorical knowledge) to return only library tags as candidates.

In practice, there could be several analogical libraries t_2 for the $base_2$ as the library t_1 for the $base_1$. Thus, we select library tags t_2 with the cosine similarity in Eq. 6.4 above a threshold *Thresh*. Take the library *nltk* (a NLP library in python) as an example. As shown in the Fig. 5.6, for *python*, our approach returns the analogical libraries such as *textblob* and *gensim*; for *java*, our approach returns the analogical libraries such as *stanford-nlp*, *opennlp*, and *gate*.

We empirically develop a guideline to set the candidate selection threshold. First, our experiment shows that a fixed threshold for all libraries often lead to an overall unsatisfactory recommendation. For example, low threshold results in too many irrelevant

candidates, while high threshold results in too few relevant candidates. This could be attributed to the word embeddings technique that learns more accurate tag representations for frequent tags than for less frequent tags, because frequent tags have more training data. As such, we find that threshold should be adjusted based on the usage frequency of tags. For less frequent tags, higher threshold should be used in order to filter out irrelevant candidates due to less accurate tag embeddings. In contrast, lower threshold could be used for frequent tags so that as many relevant candidates as possible will be selected. In practice, we can determine concrete thresholds for a dataset by sampling library tags with different usage frequencies and manually examine the recommendation results for the sampled libraries following the procedure in Section 5.4.3. The objective is to achieve a good balance of accuracy and coverage in the overall recommendation.

5.2.4.2 Refine initial results

The initial analogical-library candidates sometimes include libraries that are not for the given programming language (or mobile platform) *base₂*. For example, *beautifulsoup* is a *python* library for html parsing and web scraping. To find analogical libraries for *java* as the library *beautifulsoup* for *python*, by Eq. 6.4 we would obtain some libraries, such as *scraperwiki* (a library for ruby, python and php), *nokogiri* (a library for ruby), and *lxml* (a library for python). Although these libraries support similar features (e.g., html parsing, web scraping) to the *beautifulsoup*, they are not libraries for *java*. In our approach, we rely on the correlation between a library and a programming language (or mobile platform) (i.e., relational knowledge) to select the libraries for a given programming language. Specifically, we consider a programming language (or mobile platform) that a library has the strongest association with as the programming language (or mobile platform) that the library is implemented in. Using this relational knowledge, we can exclude libraries that are not for the given programming language (or mobile platform).

lib pairs	Extracted comparison questions
opennlp, stanford-nlp	Part of speech tagging in OpenNLP vs StanfordNLP
awt, swing	What is the difference between Swing and AWT?
bokeh, matplotlib	Plot topics with bokeh or matplotlib
junit, testng	Drawbacks of TestNG compared to JUnit?
log4net, nlog	log4net vs. Nlog
django-haystack, sphinx	django haystack or sphinx for simple search?

TABLE 5.1: Example questions about the comparison of the two analogical libraries in Stack Overflow

5.2.5 Assisting analogical-libraries comparison

Given a library and a recommended analogical library, developers are very likely interested in the comparison between different aspects of the two libraries, such as speed performance, reliability, documentation, community size, to determine whether the recommended library meets their needs. To migrate from the current library to an analogical library, developers also need to understand the process of migration and the needed efforts, for example, how to use the analogical library by referring to the experience with the current library. We develop a keyword-based matching method to extract Stack Overflow questions and sentences in Stack Overflow answers that likely assist developers in comparing analogical libraries.

5.2.5.1 Extract comparison questions

Given a library and one of its analogical libraries in the analogical-library knowledge base, we try to find Stack Overflow questions whose question title mentions the two libraries. To check if a question title mentions a library, We first lowercase both the question title and the library tag. If the library tag contains hyphen, we consider that a question mentions the library if the question contains any of the original form of the library tag, or no-hyphen form, or hyphen-to-space form (e.g., *stanford-nlp*, *stanford nlp*, *stanfordnlp*).

Furthermore, we develop several heuristic rules according to our observation of the extracted questions to remove the false positives that unlikely compare the two libraries:

lib pairs	related answer snippets
opennlp, stanford-nlp	i liked the stanford parser better than opennlp , again just looking at documents, mostly news articles
awt, swing	swing and awt both provide user interface components , however swing is built on top of awt
innodb,myisam	there are some features that are only available using myisam, like full text search, but unless you need these, i would go with innodb.
junit, testng	testng strives to be much more configurable than junit , but in the end they both work equally well.
d3.js, dc.js	moving onto dc.js ... steps : you need to load the following libraries and css files...
log4net, nlog	nlog seems to be better maintained : an incompatibility of log4net with .net4 remained unresolved in log4net for quite a long time ...
m2crypto, pyopenssl	the analog of command ... in m2crypto is: ... you can use m2crypto instead of pyopenssl with twisted
beautifulsoup, jsoup	jsoup is the java version of beautiful soup

TABLE 5.2: Example sentences in Stack Overflow answers about the comparison of the two analogical libraries

- Question title should not mention the two libraries like “lib1/lib2” format because this format usually means that the two libraries are the same or interchangeable in the question;
- Question should not be asked as “how” statements because such questions are usually about how to use the mentioned libraries;
- There are not quoted elements in the question tile because such questions usually are only about some specific elements of a library.

Table 5.1 lists some example questions about the comparison of analogical libraries extracted using the proposed keyword-matching method. Such questions provide developers some hints for comparing analogical libraries and selecting suitable ones for their tasks.

5.2.5.2 Extract comparison answer snippets

A Stack Overflow question may have several answers and/or comments which often comprise a long discussion thread to read. Sometimes developers may want more direct hints, for example, several sentences mentioning and comparing the two libraries, instead of the whole discussion thread. Furthermore, some statements about the comparison of the two libraries may appear in the answers to some questions which are not originally asked about the comparison of the mentioned libraries. Therefore, in addition to extracting comparison questions for analogical libraries, we also try to extract comparison sentences in question answers that mentions analogical libraries. Note that we do not extract comparison sentences from question body because the question asker

may not understand the two libraries very well and their statements may be misleading or wrong.

We first find all posts containing two libraries, same as the procedures for find comparison questions in last section. To guarantee the quality of the extracted sentences, we only extract sentences from answers with vote larger than 1 ($vote = upvote - downvote$). After extracting the candidate answers that mention the two libraries, we first remove all code snippets (enclosed by HTML tag `< code >`) and then split the answer texts into sentences by punctuation such as ". ", "!", ";", etc. We first select sentences that mention the two libraries, and then select sentences that mention only one library. To give developers a concise overview of a candidate answer, we take only the two sentences appearing in the answer as the representative snippet of the answer. The selection criteria for the two sentences are listed below in a descent priority:

- Find the first two sentences appearing in the answer that both mention the two libraries;
- If the answer has only one sentence mentioning the two library, select this sentence, and then select the first sentence that mentions either library;
- If the answer does not have any sentences mentioning the two libraries, then find the first two sentences in the answer, one of which mentions one library and the other sentence mentions the other library.

After finding the two sentences, we show them in the same order as they appear in the original answer. Finally we rank the candidate answers by their vote to display to the developers.

Some example comparison answer snippets can be seen in the Table 5.2. We can see that these answer snippets provide more direct hints about different aspects, usage differences and migration steps of the two libraries.

FIGURE 5.6: The screenshot of our website *SimilarTech* and the important elements in the website

5.3 Tool Support

This section describes the proof-of-concept implementation of our approach and the practice of search engine optimization so that our website can be indexed by search engines.

5.3.1 Tool description

We develop a web application with two parts. One is called *SimilarTech* (<https://graphofknowledge.appspot.com/similartech>) and the other is called *SimilarMobileTech* (<https://graphofknowledge.appspot.com/similartech/mobile.html>). Given a library name, *SimilarTech* automatically recommends its analogical libraries for different programming languages and *SimilarMobileTech* recommends analogical libraries across different mobile platforms. The current backend of *SimilarTech* and *SimilarMobileTech* is an analogical-libraries knowledge base built with the Stack Overflow data dump that contains Stack Overflow post data from July 31st, 2008 to August 16th, 2015. The backend knowledge base can be updated periodically as new data dumps are released.

The data dump that we use in the current implementation contains 9,970,064 questions and 41,856 different tags. As some infrequent or emerging tags do not have corresponding TagWiki, we collect in total 36,197 tags that have TagWiki for mining relational and categorical knowledge of tags and for learning tag embeddings. Among 36,197 tags in our dataset, 7,783 tags are categorized as library tags. We use the implementation of continuous skip-gram algorithm [74] in Word2Vec⁶ to learn tag embeddings. We set tag embedding dimension at 200.

In the current implementation, *SimilarTech* recommends analogical libraries for the top-six most frequently-asked programming languages in Stack Overflow, i.e., *java*, *javascript*, *c#*, *php*, *python* and *c++*. *SimilarMobileTech* recommends analogical libraries for the top-three most frequently-asked mobile platforms, i.e., *ios*, *android* and *windows-phone*.

Fig. 5.6 shows a screenshot of our *SimilarTech*. *SimilarMobileTech* has the same user interface design. Given a library, *SimilarTech* presents up to four libraries with the highest similarity for each programming language. The rationale is that developers would be unlikely to look through a long list of recommendations and there are usually just a few most popular libraries for each programming language. Note that listing up to four libraries is only an implementation decision, not a limitation of our approach.

Different programming languages may have different numbers of recommended analogical libraries. This is natural because some programming languages have more alternatives for a particular task, while others have less. In some cases, a programming language may not have any analogical libraries for the given library. For example, developers rarely use *javascript* for machine learning tasks. Thus, there are no well-known machine learning libraries written in *javascript*. For the machine learning library *weka*, none of the *javascript* libraries is similar enough to the *weka*. In such cases, *SimilarTech* recommends no libraries for that particular programming language. In the same vein, *SimilarMobileTech* may not recommend analogical libraries for a particular mobile platform.

⁶<https://code.google.com/p/word2vec/>

For each recommended analogical library, both *SimilarTech* and *SimilarMobileTech* show a brief definition extracted from the corresponding TagWiki. They also summarize the number of questions tagged with a library per month, and plots the metrics over time in a so-called asking trend. The asking trends of analogical libraries allow the user to easily compare the amount of the questions for each library on Stack Overflow. This information could provide hints about community size of library users and availability of online learning resources, and offer some indicators of library popularity [159].

Clicking the recommended library navigates to the analogical-library page of the clicked library. Clicking the button “Comparison” for a recommended library navigates to the comparison page between the searched library and the clicked library. The comparison page presents comparison questions and answer snippets extracted from Stack Overflow discussions, which could aid users to compare commonalities and differences of the two libraries and understand the potential migration issues.

5.3.2 Search engine optimization

Our website provides a portal to the mined analogical-library knowledge base. However, developers will not benefit from our knowledge base unless they are aware of the presence of our website and use the information in our website when they search analogical libraries. Therefore, we carry out search engine optimization (SEO) for our website so that it can be indexed and recommended by search engines. SEO is the process of maximizing the number of visitors to a particular website by ensuring that the site appears high in the search results returned by a search engine for certain types of queries⁷.

To achieve the SEO, we take three steps for our website. First, we submit all pages inside our website to several main-stream search engines such as Google, Bing and Yandex so that these search engines will crawl our site and index its content inside their database. Each page in our website contains information about a library and its analogical libraries. Second, based on the recommended analogical libraries for a library,

⁷https://en.wikipedia.org/wiki/Search_engine_optimization

FIGURE 5.7: Two example search in Google and our page ranks the first in the search results

we automatically generate a brief description for each page in our website and add this description as the title and metadata for each page. Search engines use the provided webpage title and metadata to present the webpage in the search results. Examples of webpage title and metadata in the search results can be seen in Fig. 5.7. The shown webpage title and metadata can lead web searchers to our website for further information. They can also be used as extended suggestions which may help web searchers refine their queries even without visiting our website. Third, we make our website design mobile-friendly, as such optimization can rank our site higher in search engine⁸.

After SEO, for some queries (e.g., xxx similar libraries or xxx alternative libraries), our webpages can rank quite high in the search results especially by Google search engine. Two examples can be seen in Fig. 5.7. The two queries were issued on November 17, 2016 in Singapore and Australia. Our “nltk” and “jspdf” page rank the first for the respective query. As Google adjusts the page ranking not only based on page content and links, but also click rate⁹, the high-ranking of our webpages in the search results means that many users truly click our webpages from the search results. According to the Google Analytics for our website, on average more than 2000 users around the world visit our site from their Google search results in each month. Our website usage data is limited but promising. It demonstrates the popularity and usefulness of our knowledge base and website. The usage data of our site will be further analyzed in Section 5.6.

⁸<https://webmasters.googleblog.com/2016/03/continuing-to-make-web-more-mobile.html>

⁹<https://goo.gl/nb5czF>

5.4 Accuracy Evaluation

In this section, we evaluate the mined relational and categorical knowledge of tags and the accuracy of analogical-libraries recommendations. Then we zoom-into specific cases in which our approach makes poor recommendation to understand the limitations of our approach.

5.4.1 The accuracy of tag categorization

We randomly sample 500 tags from 30,632 tags whose categories can be determined using either the NLP method or the dictionary look-up method (see Section 5.2.1). We manually examine the category of these 500 tags by reading their corresponding TagWiki. Among these 500 sampled tags, 407 (81.4%) tags are correctly labeled by our proposed methods. According to our observation, two reasons lead to the erroneous categorization. First, some tag definition sentences are complex which can lead to erroneous POS tagging results. For example, the tagWiki of the tag *pyml* states that "PyML is an interactive object oriented framework for machine learning written in Python". Our method recognizes *object* as the category because it is the first noun after the *be* verb. Second, the dictionary look-up method sometimes makes mistakes. For example, the TagWiki of the tag *honeypot* states "A trap set to detect or deflect attempts to hack a site or system". Our approach matches the *system* as the category of the *honeypot*.

As this work focuses on tags whose categories can be regarded as library, such as *library*, *framework*, *api*, *toolkit*, *wrapper*, *etc.*, we further check the correctness of these library tags in the sampled tags. Among the 500 sampled tags, there are 96 tags whose category can be regarded as library. 81 out of these 96 tags (84.4%) are correctly categorized. The accuracy of our tag categorization provides a solid basis for the analogical-libraries reasoning tasks.

5.4.2 The semantic distance of tag correlations

To evaluate the mined relational knowledge of correlated tags, we adopt the metric called “Google distance” [160, 161]. Google distance is a crowd-scale method to measure the semantic distance between a set of words by analyzing search engine data. The assumption is that the co-occurrence of a set of words in the same queries is a good indicator of the semantic distance between the words.

In this work, we use Google Trends [152] to evaluate the semantic distance of the correlated tags in the mined tag correlation graph. Google Trends is a public web service that shows how frequent a particular search-term is searched compared with the total search-volume in Google search. Given a pair of correlated tags (e.g., <java, swing>) in the tag correlation graph, we query the Google Trends with the two tags as a search term (i.e., “java swing”). Google Trends will provide the trend statistics for popular queries, and report “no enough data” for non-popular queries¹⁰.

We randomly sample 1,000 pairs of tags (i.e., tag relations) in our tag correlation graph. A small percentage of tag relations (13.1%) are not present in Google Trends (i.e., no enough data”). That is, these pairs of tags are not popular queries according to Google Trends. However, a pair of tags not present in Google Trend does not necessarily indicate wrong tag relations. First, some tags of emerging techniques (e.g., *apiary.io*) may not accumulate enough search volume on Google. Second, the difference between tagging behavior and search behavior could also result in a small percentage of tag pairs not present in Google Trend. For example, Stack Overflow users always use *javascript* and *video.js* together to tag questions, while web users search Google with *video.js* only without *javascript*.

Among the 1,000 sampled tag relations, 137 are correlations between a programming language and a library. We further check the semantic distance of these 137 library-programming-language correlations. The results show that 88.3% of these 137 correlations appear in Google Trends. Overall, the mined relational knowledge of tags can

¹⁰The detailed threshold to discriminate popular or unpopular queries is a commercial secret of Google.

accurately represent the semantic relationships between software-specific entities, including programming languages and libraries.

5.4.3 The accuracy of analogical-libraries recommendation

We first describe the procedure and metric to evaluate the accuracy of our analogical-libraries recommendation. Then, we present the evaluation results.

5.4.3.1 Evaluation procedure and metrics

We randomly sampled 100 libraries as the test cases from our analogical-library knowledge base for programming languages. As analogical-library knowledge base for mobile platforms is smaller, we randomly sampled 40 libraries as the test cases for mobile platforms. These test-case libraries support a diverse set of functionalities, such as visualization, networking, machine learning, searching, testing, and so on. 7 students are recruited (6 PhD student and 1 master student) to evaluate the accuracy of our recommendation. All the participants are majored in computer science and have at least 4-year programming experience. Each of the participants is randomly assigned 20 test-case libraries and they are asked independently judge the accuracy of our analogical-library recommendation for the assigned 20 libraries. To evaluate the impact of different kinds of knowledge for analogical-library recommendation, we ask the participants to evaluate the accuracy of four different methods: tag embeddings and relational and categorical knowledge of tags (denoted as $w2v + rc_k g$ in the following discussion), tag embeddings and relational knowledge of tags ($w2v + r_k g$), tag embeddings and categorical knowledge of tags ($w2v + c_k g$), and tag embeddings only ($w2v$).

As there is no ground truth of analogical libraries, the participants have to manually check each recommended library for a given test-case library. They examine information from library's official website, TagWiki, wikipedia, and other available online information. If the recommended library can provide comparable features as the given test-case library, the recommendation is considered as correct. Note that we do not

consider relevant libraries as correct recommendations. For example, *SimilarTech* recommends the *powermock* and *mockito* for the library *junit*. *powermock* and *mockito* are mocking framework for testing. Although *powermock* and *mockito* are relevant to the library *junit*, we do not consider them as analogical library to *junit*, because they do not provide comparable features as the given library.

Our approach is inspired by the use of word embeddings to solve analogy questions of word pairs [155]. The original word-pair analogy tasks includes two sets: semantic analogies such as *Paris – France* $\approx$? – *Spain* and syntactic analogies such as *quickly – quick* $\approx$? – *slow*. In these work-pair analogy tasks, there is only one correct answer, for example *Madrid* for *Paris – France* $\approx$? – *Spain*, and *slowly* for *quickly – quick* $\approx$? – *slow*.

In contrast, our analogical-libraries task may return several analogical libraries for a given library, as there is rarely only one solution in software engineering context. For example, for the NLP library *nlk* for *Python*, there are several comparable libraries for *Java*, such as *stanford-nlp*, *opennlp*, *gate*. Therefore, we use the Precision@k metric [162, 163] to evaluate the accuracy of analogical-libraries recommendation. Note that as the set of all analogical libraries is literally unknown, it is impossible to evaluate Recall@k.

For a given test-case library, let's assume that *SimilarTech* recommends at least one library for n ($1 \leq n \leq 6$) programming languages. Let $correct_i \bullet k$ be the number of correct recommendations in the top- k recommended libraries for a particular programming language PL_i ($1 \leq i \leq n$). The $Precision_i \bullet k$ ($k = 1, 2, 3, 4, 5$ in this evaluation¹¹) for the programming language PL_i is $correct_i \bullet k / k$. We compute the Precision@k of the overall analogical-libraries recommendation as:

$$\sum_{i=1}^n \frac{Precision_i \bullet k}{n}$$

¹¹We use a small k value because there are usually only a small number of popular analogical libraries for a given library.

FIGURE 5.8: Recommendation accuracy using the combination of different kinds of knowledge

i.e., the average of the $Precision_i \bullet k$ metrics of all the programming languages with at least one recommended library. The same evaluate metric applies for the platform-based recommendation by *SimilarMobileTech*.

5.4.3.2 Accuracy results

Fig. 5.8 illustrates the Precision@k of the four recommendation methods based on different kinds of knowledge. We can see that the tag-embeddings-only recommendation performs poorly. The recommendation accuracy by tag embedding alone is less than 30%. This is because tag sentences are short and have much noisy context information, compared with natural language sentences. Incorporating relational and categorical knowledge of tags into analogical-libraries recommendation can significantly improve the accuracy of the recommendation. Categorical knowledge of tags can boost the accuracy more than relational knowledge of tags. Incorporating both knowledge yields the best accuracy. Our results suggest that incorporating domain-specific categorical and relational knowledge with tag embeddings can enhance analogical reasoning tasks in software engineering context.

When incorporating both knowledge, for language-based recommendation, the Precision@1 is 0.81, and the Precision@5 is still reasonably high at 0.67. For mobile-platform-based recommendation, the Precision@1 is 0.72 and the Precision@5 is slightly lower, i.e., 0.59. Our results show that, given a test-case library, the top-1 library that

our approach recommends for each programming language is high likely an analogical library, and the majority of the top-5 recommended libraries for each programming language are analogical libraries. The relatively lower precision for platform-based recommendation could be attributed to the nature of mobile platforms. Unlike general programming languages which can be used for many common tasks, each mobile platform usually has many unique features that differentiate one platform from another. For example, *viewdeck* is a library for iOS apps which provides new features and enchanted view controllers. *mfi* is the Apple program that hardware developers must join to be able to manufacture and brand products as being made for iOS devices. The libraries that are highly unique for a specific mobile platform usually have no alternatives for the other mobile platforms. As such, although the recommended libraries across mobile platforms may have some similarities to a given library, they are often not analogical libraries. This results in the lower precision for platform-based recommendation.

5.4.3.3 Analysis of inaccurate recommendations

Although our approach can make accurate analogical-libraries recommendations in most cases, as the first work of this kind that combines word embeddings technique with domain-specific knowledge for analogical reasoning tasks, we would like to further investigate in which cases our approach cannot make good recommendations. This will help us, as well as other researchers and designers of similar systems, understand the limitations of our approach and address them in the future.

To that end, we investigate the test cases for which the Precision@5 metrics are below 0.2, i.e., almost all the recommended libraries for a given test-case library are incorrect. We find that such test-case libraries fall into two categories: either a full-stack framework that supports a wide range of features or a library that provides a very specific feature or support some unique features for a particular language or mobile platform.

For the first case, an example is *ruby-on-rails* (a web application framework for *Ruby*). We expect that our approach can recommend analogical framework such as *node.js* for *JavaScript*, *django* for *Python*, and *codeigniter* for *PHP*. But the recommendations by *SimilarTech* do not include any such web application frameworks. The fundamental

reason for such poor recommendations is that neural network language models assume that similar words share similar context such that word embeddings can be learned from the surrounding context. However, these full-stack frameworks can be used in very diverse context, which leads to very diverse tag sentences. As a result, in the resulting word embedding space, these frameworks and their respective programming languages do not exhibit relational similarity (or linguistic regularity) which is necessary for analogical reasoning. Thus, our approach fails to recommend analogical web application frameworks for the *ruby-on-rails*.

For the second case, examples include *jnotify* and *mako*. *jnotify* is a Java library that allow Java application to listen to file system events. *mako* is a template library providing non-XML syntax which compiles into Python modules and conceptually can be considered as an embedded Python language. Due to their very specific or language-dependent features, it is unlikely that other programming languages have comparable libraries. As illustrated in the *viewdeck* and *mfi* examples above, unique libraries for mobile platforms often result in poor analogical-libraries recommendation across mobile platforms.

To sum up, our approach is not suitable for finding analogical libraries for feature-rich, full-stack frameworks or language- or platform-dependent, unique libraries.

5.4.4 The relevance of comparison questions & answer snippets

As the comparison is about the two analogical libraries, we randomly sample 70 pairs of analogical libraries for this experiment. Each of the 7 participants (same as the last section) are randomly assigned 10 pairs of libraries. They are asked to evaluate whether or not an extracted comparison question or answer snippet is about the comparison of some aspects of a given pair of libraries. For each pair of libraries, the participants evaluate the top-5 extracted comparison questions and answer snippets. Note that some library pairs may have less than 5 extracted questions or answer snippets. For this experiment, the participants evaluate in total 315 extracted questions and 336 answer snippets for the sampled 70 pairs of analogical libraries.

FIGURE 5.9: An example analogical question and its answers in Stack Overflow. Relevant libraries are often hyperlinked in the answers.

203 (64.4%) of questions and 253 (75.3%) are marked as related to the comparison about the two analogical libraries. We further check why some questions and answer snippets are marked as irrelevant of comparison. We find that some questions are about the migration from one library to another library such as “*StructureMap to Ninject conversion*”. Although such questions or answer snippets are not related to the comparison, it could still be useful information for developers who look for analogical libraries, as they could help developers understand the migration process and avoid some potential mistakes by learning others’ experience. Some comparison-irrelevant questions/answer snippets are about how the two libraries can be complemented by each other. For example, one answer snippet of *nltk* and *stanford-nlp* is “*always refer to ... for the latest instruction on how to interface stanford nlp tools using nltk ...*”. It tells how users can interface *stanford-nlp* with *nltk*. This could be useful, because *stanford-nlp* may have better performance in some aspects but developers may still want to use *nltk* due to Python’s convenience.

5.5 Usefulness Evaluation

To demonstrate the usefulness of the proposed approach for analogical-library recommendation, we sample some questions about analogical-library recommendation in Stack Overflow, and investigate how well our recommendation can answer such questions, compared with answers provided by Stack Overflow users.

5.5.1 Experimental setup

In Stack Overflow, there are many analogical questions such as “*Is there a C++ unit testing library that is similar to NUnit?*” (in Fig. 5.1) and “*Cobertura equivalent available for C# .NET?*” (in Fig. 5.9). We define several heuristic rules (e.g, question title contains “similar library”, “alternative”, “equivalent libraries”) to collect a set of candidate analogical questions in Stack Overflow. Then we manually filter out some inappropriate questions which is not about library recommendation, and finally sample 50 questions with more than one answers as the language-based analogical questions and 20 questions with more than one answers as mobile-based analogical questions¹².

After that, we recruit 7 students (same to the Section 5.4.3.1) to extract all the recommended libraries from the answers. We find that for these analogical questions, answerers often add a hyperlink to the recommended library so that readers can access the relevant resource for more details about the library (see Fig. 5.9). Based on this observation, we ask the participants to pay more attention to such elements. To make the extracted library mentions consistent with the format of tags in Stack Overflow, we lowercase all of them and replace the space with “-”. After building the ground truth sets for the sampled analogical questions, we check how many of the ground-truth answers provided by Stack Overflow users are covered by our recommendation results for each analogical question.

5.5.2 Results

10 sample questions and answers can be seen in Table 5.3. For the 50 language-based analogical questions, on average, 71.3% libraries in answers provided by Stack Overflow users are covered by the recommended libraries using our approach. The average coverage rate is 62.3% for the 20 mobile-based analogy questions. It means that the majority of the libraries mentioned in the Stack Overflow answers can be automatically recommended by our approach.

¹²The list of sampled questions can be found at <https://graphofknowledge.appspot.com/questions>

Question	Stack Overflow answers	Our recommendation
Cobertura equivalent available for C# .NET?	ncover, opencover, partcover	opencover,visual-studio-test-runner,partcover
Alternative for PHP GD library in python	python-imaging-library	python-imaging-library, pillow, scikit-image
Modern alternative to Java XStream library?	jaxb, xmlbeans, jibx	simple-framework, castor, xmlunit, jibx
Alternative to Java3D	jogl, jMonkeyEngine	jMonkeyEngine, jogl, jzy3d, worldwind
Open source Enthought Python alternative	anaconda, pythonxy	healpy, miniconda, pythonxy, canopy
PIL ImageTk equivalent in Python 3.x	pillow	pillow, pypng, scikit-image,pythonmagick
Crypto++ equivalent in C	polarssl, openssl	botan, polarssl, cryptoapi
Java equivalent for Python NLTK	opennlp	gate, opennlp, stanford-nlp, cleartk
iAd alternative for ios ad display?	adwhirl, adsense	adwhirl, chartboost, openfeint
ASIHTTPRequest equivalent for Android?	android-async-http	android-async-http, multipartentity

TABLE 5.3: Example analogical questions and their answers from Stack Overflow and analogical libraries recommended by our method

We further explore the libraries in Stack Overflow answers but are not covered by our recommendation. We conclude two reasons for the missing. First, some libraries in Stack Overflow answers are indeed in our recommendation list (i.e., above the candidate selection threshold, such as *ncover* for the first question in Table 5.3). But in this experiment, we consider only the top 4 recommended libraries as they are what our tool currently presents, and exclude the lower-ranked recommendations. Second, some libraries in Stack Overflow answers may be helpful for the analogical library mentioned in the answers, but these libraries themselves are not analogical libraries. For example, *pandas* can help the Pythons' visualization tool *matplotlib* conveniently visualize the data, but *pandas* itself can not be regarded as an alternative to the Javascript's visualization library *d3.js*. Such auxiliary libraries will not be covered in our recommendation which results in the decrease of the coverage.

Apart from the covered libraries, our recommendation also contains some extra libraries that are not in Stack Overflow answers. According to our observation, an important reason for such extra libraries in our recommendation is the emerging new libraries that are not available at the time when the questions were asked and answered. For example, the second question in Table 5.3 was asked five years ago, but the new analogical libraries *pillow* and *scikit-image* that our approach recommends appear only about three years ago. Old posts in Stack Overflow are rarely updated with such new development in the field, which is a key issue in finding analogical libraries using these online posts. Our approach provides an alternative to recommend more up-to-date information.

5.6 Field Study

We release our website *SimilarTech* to the public in November 2015 and post this news on several programming-related websites (e.g., <http://stackapps.com/questions/6667>, <http://stackapps.com/questions/6924>). Google Analytics¹³ is embedded into our site to monitor our site traffic. Furthermore, we record the detailed page visit history in our backend server, i.e., which pages users visit, when users visit our website, and the IP address of users. We also log the users' interaction with the webpage content when they click recommended libraries, library TagWiki links, comparison button, and asking trend tabs. These detailed user behavior data allows us to gain insights into our approach and tool support which may benefit research of similar recommendation systems. In addition to the general site traffic statistics, we further investigate four research questions regarding who visits our website and what the users are interested in in our website:

- RQ1: For which libraries and libraries comparisons are the users most interested in seeking analogical libraries?
- RQ2: Do the users like to find analogical libraries within the same programming language or across different languages?
- RQ3: Do the users explore the TagWiki and asking trend that our website provides?
- RQ4: Do professional developers from major IT companies visit our site?

5.6.1 Site traffic statistics

According to the results from Google Analytics, more than 34,821 users from 168 countries visited our site, from November 11, 2015 to August 29, 2017. As shown in Fig. 5.10(a), these users on average browse 1.79 pages in each session and they browse

¹³<https://analytics.google.com/>

FIGURE 5.10: The traffic of our website from Google Analytics

in total more than 67,956 pages in 37,940 sessions¹⁴. The top 4 countries are the US (20.3%), India (9.7%), Germany (6.9%), and the UK (5.7%) (Fig. 5.10(b)) which account for 42.6% visits. 8.2% users visit our site more than once (Fig. 5.10(c)) which displays their recurring interests in our tool. The usage data of our website, albeit limited, demonstrates both the needs and the interests in analogical-libraries recommendation that our approach supports.

We note that the per-session page visit is not very high for our website. In addition, visitors averagely spend only 53 seconds in each visiting session. This can be attributed to the design rationale of our website. The users of our website are expected to have on mind specific information needs for certain analogical libraries when they visit our website. Our website is designed to provide the users with a concise summary of the information they may need to find analogical libraries. Then, the users may click library TagWiki or extracted comparison questions and answer snippets to obtain more information about the recommended library in Stack Overflow. Or they may leave our website to search the recommended library for more information. As a result, we do not expect a high per-session page visit or long time in each session. In fact, through search-engine optimization, the users may obtain the key information they need (i.e., the name of the analogical libraries) from the webpage metadata displayed in the search results. They may then search the library names directly, without visiting our website. Even in this scenario, our website still fulfills its design goal to assist developers' analogical-library search.

¹⁴As most search engine robots do not activate javascript, robot traffic is not counted in Google Analytics [164].

libraries within the same language or across different languages by analyzing the logs of how users search libraries and view library information in our website.

In our website, users can enter a library name in the search box, and the website shows the analogical-library page for the searched library. On an analogical-library webpage, users can click a recommended library which brings the users to the analogical-library page for the clicked library. Users can also click “Learn more” to view the TagWiki of a recommended library. We collect the sequence of the user’s actions (excluding the visit of the home page) in our website during a visit session.

As we want to analyze the users’ interests in analogical libraries within or across languages, sequences with only one action are ignored. From the 11465 sequences that contain at least two actions, we extract pairs of consecutive actions that involve a library searched followed by an analogical library searched/viewed. We obtain 15039 such pairs. For each pair of libraries, we check the language of the libraries. Among 15039 pairs, 12721 pairs of libraries are in the same language, and the rest 2318 pairs of libraries are across different languages.

The users’ search and browsing behavior in our website seems to suggest that the users are more interested in analogical libraries within the same language than across different languages. Two reasons may account for such results. First, we place the same-language analogical libraries in the first row in the web page. This may attract more clicks. Second, many developers are most familiar with one programming language. Thus, they may prefer analogical libraries in the same language over using some libraries in another language they are not good at.

We further analyze the potential migration pattern of developers when they are interested in analogical libraries across different languages, i.e., developers want to find analogical libraries from which language to the other. Table 5.6.3 displays the results of the language migration matrix. We can see that java and c# developers more likely search for analogical libraries in other languages, compared with developers of other languages. On the other hand, *python* seems to attract developers from other languages to change to use libraries in *python*. This could be because *python* has many libraries

<div>target</div> <div>source</div>	java	javascript	python	c#	c++	php	sum
java		117	226	234	94	38	709
javascript	63		94	87	41	52	337
python	107	44		84	81	26	342
c#	150	89	101		110	26	476
c++	52	23	148	59		11	293
php	24	33	57	35	12		161
sum	396	306	626	499	338	153	

TABLE 5.4: Language migration statistics

that are good alternatives for libraries in other languages and also maybe due to its popularity in deep learning. For *php*, it seems that PHP developers do not often change to use libraries in other languages, and developers of other languages do not often change to use PHP libraries either.

It is important to note that this analysis could be biased for two factors. First, users may simply view the information on an analogical-library page without clicking any recommended libraries and/or their TagWikis. In fact, we have about 71.1% action sequences containing only one action. In such cases, users may still have interests in some information in the webpage, for example, view asking trend, highlight some words in the TagWiki snippets while reading them, copy some keywords for their further search. However, we do not have clear signals about what they may be interested in. Second, it is very likely that after finding some hints for analogical libraries in our website, users leave our website to google the libraries for more details. We cannot collect any behavior data (like libraries searched and webpages read) outside our website.

5.6.4 Usefulness of tagWiki and asking trend (RQ3)

Our web application does not just provide the names of analogical libraries, but also provides a brief description of the recommended library and a summary of asking trends of the recommended libraries. We collect the users' interaction with the provided information in order to understand whether they are useful or not. The behavior tracking component was deployed in July 3, 2016. The analysis below is based on the data collected from July 3, 2016 to Aug 29, 2017.

In the last 14 months, apart from the homepage, 42,127 library pages in our websites are visited. Among these web pages, users access the TagWiki of the searched library, asking trend tabs, the TagWiki for the recommended libraries, and/or the library comparison in 15,730 pages (37.3%). Users access the TagWikis of some recommended libraries in 2,936 pages. Users click asking trend tabs for different languages in 2,224 pages. It indicates that some users are not only interested in knowing the name of analogical libraries, but also want to know more details about our recommendations. Surprisingly, users click the TagWiki of the searched library in 11,314 pages. This could be because the design of our website which may mislead the users to click the link of a recommended library when they only want to read the TagWiki of the library. Within 1,522 pages, the user clicked the comparison button to compare the searched library with the recommended libraries. In the current design, clicking the link of a recommended library brings the users to the analogical-library page for the clicked library, while the users must click “Learn more” for a library to access its TagWiki. When the users find that clicking the link of the recommended library does not lead them to the TagWiki, they may then click “Learn more” for the clicked library on its analogical-library page to access its TagWiki. We will further improve our website design to make it more user-friendly.

Although many users do not explicitly click any elements in the web pages they visit, it does not mean that they do not get the information they need. It is likely that the users find the information they need from the TagWiki snippets and the asking trends that is already presented in the webpage. For example, the web page displays the asking trends of the analogical libraries in the same programming language as the searched library by default. If the users are only interested in analogical libraries in the same programming language as the searched library (according to the analysis in Section 5.6.3, this is likely the case), they do not need to click any asking trend tabs to get the trend information they need. Furthermore, it is impossible to track how the users use the information from our website once they leave our website. Therefore, our analysis provides a conservative estimate of how useful the information our webpage provides could be.

Company	#IPs
Google Inc.	218
Amazon.com, Inc.	207
Microsoft Corporation	98
Alibaba (China) Technology Co., Ltd.	73
Cisco Systems, Inc.	42
Oracle Corporation	41
Apple Inc.	37
Hewlett-Packard Company	35
Intel Corporation	32
Facebook, Inc.	19

TABLE 5.5: The number of visits from big companies

Metrics	Company	Rest
returning visitors	22.1%	8.2%
pages per session	1.47	1.79

TABLE 5.6: The comparison of visits between developers from big companies and the rest

5.6.5 Real developers' visits (RQ4)

For each visitor of our website, we record their IP address. Given the IP address, we can find their Internet Service Provider using the ipinfo.io service¹⁵. From the ISP, we can know the organization to which the IP address belongs. Table 5.6.5 lists several IT companies whose IP addresses visit our website frequently. As the visits to our website only last several web pages, we rule out the possibility of the web crawler. In addition, as our website is related to programming, we assume the visits from these companies are from their developers.

We further explore the visit logs of the users from the listed IT companies, and compare their behaviors with that of other users. We have in total 801 users from the listed IT companies, and these users browse 1440 pages in 979 sessions. Table 5.6.5 shows the behavior difference between the IT company users and the other users. It is more likely that the company users will revisit our website (22.1%) than the other users (8.2%). But the company users visit fewer pages (1.47) in each session than the other users (1.79). This could be because these company users are more experienced and have specific information needs on mind. Therefore, they do not need to explore the recommended analogical libraries in our website.

¹⁵<http://ipinfo.io/>

We then analyze their clicks behaviors to conclude their migration patterns. Among all 221 sessions with more than one queries, we find that most of users (90%) are interested in the similar libraries with in the same language, and the most popular languages are *Java*, *Javascript* which are widely used in the industry. The most popular library migration is the *jsPDF*¹⁶ to *PDF.js*¹⁷. For the cross-language clicks, the most frequent migration is the *C++* to *Python*, *Java* to *Python* and *Java* to *Javascript* which indicates their interest from objected-oriented programming language to the scripting languages. But note that as the log size is too small, it may not represent the real trends of the industry.

5.7 Conclusion

Third-party libraries assist developers in finishing software engineering tasks more efficiently without the need to reinvent the wheels. However, due to many reasons such as lack of active maintenance of the libraries being used or language migration, developers often need to find some alternative and comparable libraries to replace the libraries they are already familiar with. Although developers can find useful information in community-curated list, blogs and Q&A posts on the Web, the information is likely to require tedious and time-consuming browsing and aggregation, or is likely to be out of date to mislead developers especially the novice.

In this chapter, we propose an automated technique to recommend analogical libraries. We adopt the cutting-edge deep learning method in NLP applications (also known as word embeddings) to the software engineering data. We further enhance the original word embedding technique with software-engineering domain knowledge to better answer analogy questions in software engineering context. The extracted analogical relationships among third-party libraries enrich our knowledge graph so that given a library, our knowledge graph can recommend several most salient analogical libraries in different programming languages or different mobile platforms.

¹⁶<https://github.com/MrRio/jsPDF>

¹⁷<https://mozilla.github.io/pdf.js/>

Chapter 6

Unsupervised Software-Specific Morphological Forms Inference from Informal Discussions

6.1 Introduction

We have built a knowledge graph in Chapter 4 and enrich the relationships in Chapter 5. But developers queries are very likely to be informal which may lead to mismatch of keywords in their queries and entities in the knowledge graph, leading to the bad search results. As informal discussions are contributed by millions of users with very diverse technical and linguistic background, the same concept is often mentioned in many *morphological forms, including abbreviations, synonyms and misspellings*, intentionally or accidentally [99]. Fig. 6.1 shows three Stack Overflow posts that discuss the slash issue of regular expression when parsing JavaScript. These three posts are

FIGURE 6.1: The morphological forms of “regular expression” (blue) and “javascript” (red) in three Stack Overflow posts

TABLE 6.1: Morphological forms of *visual c++*

Term	Frequency	Annotation
visual c++	10,294	Standard
msvc	8,477	abbreviation
vc++	7,154	abbreviation
microsoft visual c++	1,826	synonym
ms vc++	295	abbreviation
visual-c++	110	synonym

marked as duplicate posts by the Stack Overflow community, because they discuss the same programming issue. That is, the three posts are considered as semantically equivalent. However, when mentioning regular expression and JavaScript, the three different users use many morphological forms (e.g., *regex*, *RegExp*, *regexes*), and even the same user uses various forms in the same post (e.g., *JS*, *JavaScript*). As another example, Table 6.1 summarizes the frequencies of various morphological forms of *visual c++* in Stack Overflow discussions. Note that there are many morphological forms for the same concept and some forms are used as frequently as the standard one.

The wide presence of morphological forms of the same concept in informal discussions poses a serious challenge to informal retrieval. For example, for the query “slash in regular expressions Javascript”, some posts in Fig. 6.1 may not be retrieved due to the morphological forms of *JavaScript* and *regular expression* that are used in the posts, even though the three posts are semantically equivalent. It also negatively influences the robustness of the entity search based on knowledge graph in Chapter 4 as each entity may have different synonyms and abbreviations. Overlooking the close relationships between various morphological forms of the same concept may also accentuate data sparsity problems in applying NLP techniques to mining programming knowledge in informal discussions, which could negatively affect the performance of the NLP techniques.

In the NLP domain, a recent trend has seen proposals that deal with morphology using word embeddings and neural networks [165–167]. A recent work by Soricut and Och [168] exploits the relational regularities exhibited by word embeddings (e.g., car to cars, dog to dogs) to model prefix- and suffix-based morphological rules and transformations. However, these morphology learning techniques in the NLP domain consider only morphological relations drawn out of linguistic regularities of natural language

(e.g., prefix and suffix). As shown in Fig. 6.1 and Table 6.1, morphological forms of software-specific terms found in informal discussions do not always follow linguistic regularities of natural language, e.g., *JS* and *Javascript*, *RegExp* and *regular expressions*.

In the software engineering domain, the impact of consistent vocabulary on the application of NLP-based techniques to source code and software documentation has long been recognized [87–91]. The focus has been on expanding identifiers that contain abbreviations and acronyms. The proposed solutions are predominantly lexically-based approaches, for example, based on common naming conventions in software engineering like camel case, or use string edit distance to measure the similarity between an abbreviation and its potential expansions. But lexical rules are often unreliable. For example, both *open cv* and *opencv* are lexically similar to *opencv*. However, *opencv* is a library for parsing *csv* files which is totally irrelevant to *opencv* (a computer vision library). To improve the results, most of these approaches resort to external resources (e.g., English dictionary, dictionary of IT term and known abbreviations) which are often difficult to build and maintain, especially domain-specific ones. Some approaches [63, 94, 95] exploit word frequencies in word co-occurrence data to rank abbreviation expansions, but none of them exploit semantic relatedness of words.

In this chapter, we propose an automatic approach for inferring morphological forms of software-specific terms in a large corpus of informal software engineering text, and these morphological forms are incorporated into the knowledge graph in Chapter 4 to make the corresponding entity-search more robust. Our approach first contrasts software engineering text (e.g., Stack Overflow discussions) against general text (e.g., Wikipedia documents) to derive a vocabulary of software-specific terms used in software engineering text. It then combines the latest development of word embeddings in the NLP domain and the domain-specific lexical rules developed in the software engineering domain. As such, we can infer morphological forms of software-specific terms that not only obey lexical rules but also are semantically close to each other. Based on the graph of the morphological relations between pairs of terms, our approach finds groups of morphological forms, each expressing a distinct concept (see Fig. 6.5 for examples), similar to the notion of synset of WordNet [85].

Compared with several community-curated lists of IT terms, abbreviations and synonyms, our approach automatically infers software-specific terms and morphological forms that are up-to-date and actively used in Stack Overflow. In contrast, community-curated lists contain many out-of-date terms and rarely-used morphological forms. This result demonstrates the needs for and the advantage of an approach like ours for automatic software-specific morphological form inference. Manual examination of randomly sampled 1,200 abbreviations and synonyms confirms the high accuracy (81.9%) of our approach. To demonstrate the usefulness of the inferred morphological forms of software-specific terms for information retrieval, we use the inferred morphological forms to normalize question text and question metadata (i.e., tags) from both Stack Overflow and CodeProject. Our results show that our morphological forms can better improve the consistency between question text and question metadata, compared with Porter stemming [169] and WordNet-based lemmatization [170] that are commonly used for English text normalization. Furthermore, the results show the generality of our morphological forms across different software engineering corpus.

6.2 Approach

As shown in Fig. 6.2, the input to our approach is only a software-specific corpus (e.g., Stack Overflow text) and a general corpus (e.g., Wikipedia text). Our approach includes six main steps: 1) text cleaning and phrase detection, 2) identifying software-specific vocabulary by contrasting software-specific and general corpus, 3) learning term semantics by word embedding techniques (e.g., continuous skip-gram model [74]), 4) extracting semantically related terms as candidates of morphological forms, 5) discriminating abbreviations and synonyms from the list of morphological-form candidates, and 6) based on a graph of morphological relations, grouping morphological forms of software-specific terms. The output of our approach is a thesaurus of software-specific terms and their morphological forms (called *SEthesaurus*).

FIGURE 6.2: The overview of our approach

6.2.1 Dataset

Our approach takes a software-specific corpus of plain text and a general corpus of plain text as inputs. No other external resources are required. Software specific corpus can be crawled from domain-specific websites, such as Stack Overflow, CodeProject, W3School, MSDN. As we are interested in discovering morphological forms in informal discussions, as well as considering the popularity of the website and the volume of the data, we choose Stack Overflow text as software-specific corpus in this work. General corpus can be crawled from domain-agnostic websites, such as Wikipedia, Quora, Baidu Zhidao, which cover a diverse set of domains. Considering the quality and the public availability of the data, we choose Wikipedia text as general corpus in this work. Wikipedia text is also adopted as general corpus in other NLP work [171]. It is important to note that our data-analysis approach is not limited to Stack Overflow and Wikipedia data.

Raw dataset: In this work, the Stack Overflow data dump [172] that we use contains 9,970,064 questions and 16,502,856 answers from July 2008 to August 2015. We collect the title and body content of all the questions and answers as the software-specific

corpus. The Wikipedia data dump [173] includes 5,044,130 articles before December 2015. We collect the page content of all the articles as the general corpus.

6.2.2 Preprocessing input corpora

6.2.2.1 Text cleaning

As both datasets are from websites, we follow the text cleaning steps commonly-used for preprocessing web content [174, 175]. We preserve textual content but remove HTML tags. For Wikipedia data, we remove all references from page content. For Stack Overflow, we remove long code snippets in `<pre><code>` in the posts, but not short code elements in `<code>` in natural language sentences. We use our software-specific tokenizer [158] to tokenize the sentences. This tokenizer preserves the integrity of code-like tokens and the sentence structure. For example, it treats `pandas.DataFrame.apply()` as a single token, instead of a sequence of 7 tokens, i.e., `pandas . DataFrame . apply ( )`.

6.2.2.2 Phrase detection

A significant limitation of prior techniques is that they consider only single word. However, many software engineering terms are composed of several words such as *ruby on rails*, *visual studio* and *depth first search*. These multi-words phrases must be recognized and treated as a whole in data analysis.

We adopt a simple data-driven and memory-efficient approach [74] to detect multi-words phrases in the text. In this approach, phrases are formed iteratively based on the unigram and bigram counts, using

$$score(w_i, w_{i+1}) = \frac{count(w_i w_{i+1}) - \delta}{count(w_i) \times count(w_{i+1})} \quad (6.1)$$

The w_i and w_{i+1} are two consecutive words. δ is a discounting coefficient to prevent phrases consisting of two infrequent words to be formed. That is, the two consecutive words will not form a bigram phrase if they appear as a phrase less than δ times in the

corpus. In this work, we experimentally set δ at 10 and the threshold for *score* at 15 to achieve a good balance between the coverage and accuracy of the detected multi-words phrases.

Our method can find bigram phrases that appear frequently enough in the text compared with the frequency of each unigram, such as *sql server*. But the bigram phrases like *this is* will not be formed because each unigram also appear very frequently separately in the text. Once the bigram phrases are formed, we repeat the process to detect trigram and fourgram phrases. In this work, we stop at fourgram phrases, but the approach can be extended to longer phrases.

Corpus summary: After text cleaning and phrase detection, we obtain a software-specific corpus from the Stack Overflow data dump, which has 8,125,944 unique terms (including single words and multi-words phrases) and 1,757,436,186 tokens (a token is a mention of a term). We obtain a general corpus from the Wikipedia data dump, which has 26,639,445 unique terms and 2,356,736,103 tokens.

6.2.3 Building software-specific vocabulary

Inspired by Park et al’s work [171], we identify software-specific terms by contrasting the term frequency of a term in the software specific corpus compared with its frequency in the general corpus. Specially, we measure the term’s *domain specificity* based on the equation:

$$domainspecificity(t) = \frac{p_d(t)}{p_g(t)} = \frac{\frac{c_d(t)}{N_d}}{\frac{c_g(t)}{N_g}} \quad (6.2)$$

where d and g represents software-specific and general corpus respectively, and $p_d(t)$ and $p_g(t)$ is the probability of the term t in the two corpuses respectively. The probability of a term in a corpus is calculated by dividing the term frequency by the total number of tokens N in the corpus. The underlying intuition is that terms that appear frequently in software-specific corpus but infrequently in general corpus are software-specific terms. In this work, we experimentally set 10 as the threshold for *domainspecificity* to discriminate software-specific terms.

FIGURE 6.3: The continuous skip-gram model. It predicts the surrounding words given the center word.

We observe that some terms that developers commonly use on Stack Overflow bear little domain-specific meaning. For example, *i* is frequently used as variable in loop. Developers also frequently mention some numeric metrics, such as *1 sec* and *10mb*. As these terms do not represent any domain-specific concepts in natural language discussions, we set stop-term rules to exclude such meaningless terms, for example, excluding terms beginning with number or special punctuations like ***, *+* and *>*, excluding terms with only one letter (*c* and *r* are preserved as they are programming languages).

6.2.4 Learning term semantics

To capture the semantics of software-specific terms, we adopt the continuous skip-gram model [74, 176] which is the state-of-the-art algorithm for learning distributed word vector representations (or word embeddings) using a neural network model. The underlying intuition of the algorithm is that words of similar meaning would appear in similar context. Therefore, the representation of each word can be defined on the words it frequently co-occurs with.

As illustrated in Figure 6.3, the objective of the continuous skip-gram model is to learn the word representation of each word that is good at predicting the surrounding words in the sentence. Formally, given a training sentence of K words $w_1, w_2, \dots, w_K$, the objective of the continuous skip-gram model is to maximize the following average log probability:

$$L = \frac{1}{K} \sum_{k=1}^K \sum_{j=-N, j \neq 0}^N \log p(w_{k+j} | w_k) \quad (6.3)$$

where w_k is the central word in a sliding window of the size $2N + 1$ over the sentence, w_{k+j} is the context word surrounding w_k within the sliding window. Our approach trains the continuous skip-gram model using the software-specific corpus obtained in Section 6.2.2. We set the sliding window size N at 5 in this work. That is, the sliding window contains 10 surrounding terms as the context terms for a given term in the sentence.

The probability $p(w_{k+j} \mid w_k)$ in Eq. 6.3 can be formulated as a log-linear softmax function which can be efficiently solved by the negative sampling method [74]. After the iterative feed-forward and back propagation, the training process finally converges, and each term obtains a low-dimensional real-valued vector (i.e., word embedding) in the resulting vector space. Following the experiments in our previous work to learn word embeddings from Stack Overflow corpus [98], we set the dimension of word embeddings at 200.

Stack Overflow is time-sensitive due to the evolution of technology landscape [159]. New terms emerge all the time, and existing term usage also changes over time. Word embedding is not good at encoding less frequent terms. If trained using the entire data, semantics of no-longer-actively-used or newly-appearing terms may not be well captured. To mitigate this issue, we split the Stack Overflow corpus into M bulks of data evenly ($M = 11$ in this work, about 2.4 million posts per bulk). For each bulk of data b_i ($1 \leq i \leq M$), we apply the continuous skip-gram model to the data and obtain a corresponding vector space V_i .

6.2.5 Extracting semantically related terms

For each software-specific term t in the software-specific vocabulary, if the term t is in the vector space V_i ($1 \leq i \leq M$), we find a list of semantically related terms whose term vectors $v(w)$ are most similar to the vector $v(t)$ in the vector space using the following equation:

$$\operatorname{argmax}_{w \in A_{V_i}} \cos(v(w), v(t)) = \operatorname{argmax}_{w \in A_{V_i}} \frac{v(w) \cdot v(t)}{\|v(w)\| \|v(t)\|} \quad (6.4)$$

FIGURE 6.4: An illustration of term representations in the word embedding space, two-dimensional projection using PCA.

where A is the set of all terms in the vector space V_i excluding the term t , and $\cos((v(w), v(t)))$ is the cosine similarity of the two vectors.

For a term $t \in V_i$, we select the top-20 most similar terms in the vector space V_i as the candidate semantically related terms. As we split the whole corpus into M bulks, we obtain M vector spaces. Let X be a set of vector spaces that contains the term t ($1 \leq |X| \leq M$). Therefore, we obtain $|X|$ candidate lists for the term t . These $|X|$ candidate lists could overlap. We merge the $|X|$ candidate lists into one list and re-rank the candidate terms w based on the equation:

$$semsim(w, t) = \frac{\sum_{V_i \in Y} \cos_{w \in V_i}(v(w), v(t))}{|Y|} \times \log_{|X|} |Y| \quad (6.5)$$

where Y is a set of vector spaces that contain both the candidate term w and the term t ($1 \leq |Y| \leq |X|$). The semantic similarity of the candidate term w to the term t is proportional to the two components: first, the average of the w 's cosine similarity with the term t in the $|Y|$ vector spaces, and second, the logarithm of $|Y|$ on the base $|X|$. In practice, we add 1 to $|X|$ so that the log base is not 1 and add 1 to $|Y|$ so that the log will not be 0 when $|Y| = 1$. We take the logarithm of $|Y|$ on the base $|X|$ so

that the two components in the equation have comparable contributions. This equation lessens the importance of some terms w which may only be highly related to the term t in a small number of vector spaces. Meanwhile, the less frequently used terms will not be overwhelmed by the more frequently used terms. We select the top-20 candidate terms in the reranked list as the semantically related terms for the term t .

Fig. 6.4 illustrates the set of semantically related terms for the three terms *angularjs*, *mac os x* and *natural language processing*. In the Figure, for the sake of clarity, we list only the top six most similar terms for the three terms respectively. These terms are projected into a two-dimensional vector space using Principal Component Analysis (PCA) [177], a technique commonly used to visualize high-dimensional vectors. We can see that semantically related terms are close to each other in the vector space. Furthermore, we can observe three kinds of relations between semantically related terms, 1) synonyms, e.g., (*angular*, *angular.js*), (*mac os x*, *macosx*); 2) abbreviations, e.g., (*natural language processing*, *nlp*); and 3) general relatedness, e.g., (*max osx*, *ubuntu linux*), (*angularjs*, *ember*), and (*nlp*, *data mining*). In this work, we focus on abbreviations and synonyms (referred to as morphological forms of a term in this work) among semantically related terms. Generally related terms could be also useful for recommendation systems, they are applied to build a software-specific dictionary in Section 6.5.4.

6.2.6 Discriminating synonyms & abbreviations

We now explain the lexical rules and the string edit distance we use to discriminate synonyms and abbreviations of a term from its semantically related terms.

6.2.6.1 Discriminating morphological synonyms

In this work, we define synonyms as pairs of morphological similar terms. Some morphological-synonyms can be determined by stemming, such as (*object*, *objects*), (*rebase*, *rebasing*), but many other cannot, such as (*objective-c*, *objective c*), (*mac os x*, *macosx*), (*algorithm*, *algoritm (a misspelling)*), (*angular*, *angularjs*). We observe that morphological-synonyms among semantically related terms usually can be transformed

from one term to another by a small number of string edits. Therefore, given a term t and a semantically related term w , we use string edit distance to determine whether the two terms are morphological-synonyms.

Levenshtein distance [178] is often used to compute the string edit distance i.e., the minimum number of single-character edits (insertions, deletions or substitutions) required to transform one word into another. In this work, we use an enhanced string edit distance, Damerau-Levenshtein distance [179] (DL distance) to compute the minimum number of single-character edits (insert, delete, substitute, and transposition) required to transform one term to another. DL distance enhances the Levenshtein distance [178] with the transposition of the two adjacent characters such as *false* and *flase*. Such character transpositions are a common source of misspellings. DL distance can more reliably detect such misspellings than the Levenshtein distance.

The absolute DL distance cannot be directly adopted for measurement. For example, the DL distance between *subdomain* and *sub-domains* and the DL distance between *jar* and *jsp* are both 2. The pair (*subdomain*, *sub-domains*) is morphological synonyms, while the pair (*jar*, *jsp*) is not. Therefore, we take into consideration both the absolute distance and the relative similarity between two term. For the absolute distance, the DL distance of the two synonyms must not be greater than 4, for example, the pair (*dispatcher.begininvoke*, *dispatcher.invoke*) will not be regarded as synonyms because the absolute DL distance between the two terms is 5.

For the relative similarity, we normalize the DL distance according to the maximum length of the two terms by:

$$similarity_{morph}(t, w) = 1 - \frac{DLdistance(t, w)}{\max(len(t), len(w))} \quad (6.6)$$

The relative similarity indicates that the different parts of the two synonyms should be relatively small compared with the same parts of the two terms. In this work, we set the relative similarity threshold at $\frac{1}{3}$. As a result, the pair (*subdomain*, *sub-domains*) will be recognized as synonyms, but the pair (*jar*, *jsp*) will not, because the first pair is relatively similar enough, but the second pair is not.

TABLE 6.2: Example Abbreviations and Synonyms

RepTerm	Abbreviations	Synonyms
applicationcache	appcache	application cache
android-query	aquery	android query, androidquery
codeigniter	ci	codeingiter, codeignitor
algorithm	algo, algos	algoritms, algorithm
blackberry 10	bb10, bb 10	blackberry10

6.2.6.2 Discriminating abbreviations

If a semantically related term w does not satisfy the requirement of being a synonym of a given term t , we further check whether it is an abbreviation of the given term. We consider the semantically related term w as an abbreviation of the term t if they satisfy the following heuristics-based lexical rules. Similar rules are used to expand to identify abbreviations [87, 90].

- The characters of the term w must be in the same order as they appear in the term t , such as (*pypi*, ***python*** *package index*), (*amq*, ***activemq***);
- The length of the term w must be shorter than that of the term t ;
- If there are digits in the term w , there must be the same digits in the term t . For example, *vs2010* is regarded as an abbreviation of *visual studio 2010*, but *vs* is not regarded as an abbreviation of *visual studio 2010*;
- The term w should not be the abbreviation of only some words in a multi-words phrase. For example, *cmd* is regarded as the abbreviation of *command*, but not as the abbreviation of *command line*.

It is important to note that we discriminate morphological synonyms and abbreviations from highly semantically related terms established by the terms' word embeddings. The above edit distance and lexical rules alone cannot reliably detect morphological synonyms and abbreviations without considering semantic relatedness between terms. For example, according to the above lexical rules, *ie* can be regarded as an abbreviation of ***view***. However, once considering semantic similarity, the term *ie* is not semantically related to the term *view*. Thus, *ie* will not even be an abbreviation candidate for *view*.

FIGURE 6.5: The synonym graph: the edge represents the synonym relationship and a connected component represents a group of synonyms (zoom-in to view the details)

Similarly, by solely DL distance, the terms (*opencv*, *opencsv*) will be regarded as synonyms. However, in our approach the two terms are not semantically related, and thus neither of them will be considered as synonym candidate for each other.

6.2.7 Grouping morphological synonyms

We identify synonyms for each term in our software-specific vocabulary. It is likely that we obtain separate but overlapping sets of synonyms for different terms. For example, for the term *timed-out*, we obtain {*timedout*, *timed out*}, while for the term *timeout*, we obtain {*timeout*, *timeouts*, *timed out*}. Note that the term *timed out* is in the two synonym sets. We group such overlapping sets of morphological synonyms for different terms into one set of morphological synonyms in which each pair of terms can be regarded as morphological synonyms.

To group separate but overlapping sets of morphological synonyms, we first build a graph of morphological synonyms based on the synonym relations between terms. Then, we find connected components in the graph as groups of morphological synonyms. Each pair of terms in a group is considered as synonyms. Figure 6.5 shows some examples¹.

¹For multi-words phrases (e.g., *git rebase*), we replace space with “_” for the visualization clarity.

TABLE 6.3: The coverage of terms in SOTag and CPtag datasets

Dataset	#Term	#CoveredTerm	Coverage
SOTag	21950	15385	70.1%
CPtag	2391	1893	79.2%

For example, the term *timeout* is regarded as a synonym of *timedout* via the term *timed out*.

Considering all terms in a connected component as mutual synonyms, we essentially consider each group of morphological synonyms as a distinct concept. We select the term in the group with the highest usage frequency in Stack Overflow as the representative term of the concept. For each group of morphological synonyms (i.e., each concept), we merge the list of abbreviations of the terms in the group into a list of abbreviations for the group. Table 6.2 presents some examples of the representative terms and their abbreviations and synonyms identified by our approach.

6.3 Evaluation of Our Thesaurus

In this section, we evaluate the coverage of software-specific vocabulary and the coverage of abbreviations and synonyms in our thesaurus *SEthesaurus* against several community-created lists of computing-related terms, abbreviations and synonyms. We also manually examine the correctness of the identified abbreviations and synonyms. The evaluation confirms the effectiveness of our approach, meanwhile reveals potential enhancements of our current approach.

6.3.1 The coverage of software-specific vocabulary

Our thesaurus contains 52,645 software-specific terms. To confirm whether our thesaurus covers a good range of software specific terms, we compare the software-specific vocabulary of our thesaurus against the three community-curated sets of software-specific terms.

In Stack Overflow, each question is tagged with up to five terms that describe the programming techniques and concepts of the question. These tags can be regarded as software-specific terms. Considering the power law distribution of tag usage, we consider tags that are used at least 30 times to avoid rare terms, and we collect in total 21,950 tags (as of August 2015) from over 9-millions questions to check if these tags are in our vocabulary. In addition, we also collect 2,391 tags from 263-thousands questions in the other programming Q&A site, CodeProject. For brevity, we refer these two datasets as *SOTag* and *CPtag*.

Our software-specific vocabulary covers 70.1% terms in the *SOTag* dataset, and 79.2% terms in the *CPtag* dataset. By observing the Stack Overflow and CodeProject tags, we further find three reasons why some tags are not covered by our software-specific vocabulary. We explain our observations using Stack Overflow dataset.

First, some tags contain four or more words, and many of them contain version number such as *google-maps-api-3* and *ruby-on-rails-3.2*. However, people often do not mention version numbers when mentioning a technique in discussions. Therefore, our vocabulary may not contain a specific version of a technique, but it usually contain the general term for the technique, such as *google maps api*, *ruby on rails*. The coverage for the tags with 4 or more words is low (about 20%). However, for the tags with 3 or less words, the coverage becomes much higher. For tags that are used more than 1000 times, the coverage by our vocabulary can reach 90% or higher. But for tags that are used less than 100 times, the coverage is only about 54.6%. Note that although less frequently-used tags (30-1000 times) account for 86% of the tags, their total times of usage in Stack Overflow account for only 6.1% of the total tag usage. Therefore, the impact of missing some less frequently used tags (especially those used 30-100 times) on NLP tasks like information retrieval is minor. Third, some tags are artificial terms for question tagging, such as *android-asynctask* and *django-views*, but these terms are rarely used in discussion text.

6.3.2 Abbreviation coverage

Our approach finds 4,773 abbreviations for 4,234 terms (one term may have several abbreviations) from Stack Overflow corpus. In Wikipedia, there is a list of computing and IT abbreviations [180]. The list contains 1,292 full names and each full name has one abbreviation, except for *regular expression* which has two abbreviations. 855 of these 1,292 full names are in our vocabulary. For those 437 full names that are not in our vocabulary, they are either long phrases (e.g., *context and dependency injection*, *advanced data communications control procedures*) or related to other domains such as communication (e.g., *atm adaptation layer*, *advanced research projects agency*), and thus are not mentioned frequently enough in Stack Overflow for our approach to identify them as software-specific terms.

We use the 855 full names and their abbreviations as ground truth to examine the coverage of the identified abbreviations in our thesaurus. 751 of these 855 full names have abbreviations in our thesaurus, and the abbreviations of 739 out of the 751 terms are in the ground truth. That is, the accuracy is 86%. According to our observation, two reasons result in the missing abbreviations. First, there are some unusual abbreviations in the Wikipedia list which we believe developers more like to use full names instead of the abbreviations, e.g., *access time* instead of *at*. Second, there are limitations with our abbreviation inference heuristics which cannot find abbreviations with unique string transformations, such as *i18n* for *internationalization*, *xss* for *cross-site scripting*, and *w3c* for *world wide web consortium*. In fact, our approach identifies these abbreviations as semantically related to their full names. However, due to their unique string transformation, general lexical rules cannot determine them as abbreviations.

Compared with the Wikipedia abbreviation list, our dictionary contains much more software-specific terms and more abbreviations, for example, *abc* for *abstract base class*, *sso* for *single sign-on*. Furthermore, our approach can capture multiple abbreviations for a term. For example, our approach finds 7 abbreviations (*regex*, *reg exp*, *regexps*, *regexes*, *regexp*, *regexs*, *reg-ex*) for *regular expression* in the Stack Overflow text, while the Wikipedia list include only two of these 7 abbreviations.

6.3.3 Synonym coverage

Our approach identifies 14,006 synonym groups which contain 38,104 morphological terms². To examine the coverage and accuracy of the identified synonyms, we compare our results against the Stack Overflow tag synonyms. Stack Overflow users collaboratively maintain a list of tag synonym pairs. By August 2015, there are 3,231 community-approved synonym pairs [181]. Each pair has a synonym tag and a master tag. We take these tag synonym pairs as the ground truth. According to Stack Overflow tag naming convention, multi-words in a tag are concatenated by “-”, while in plain text, users more likely write them with spaces. Thus, we replace “-” in the tag with space for this comparison, for example, the tag *visual-studio* will be transformed into *visual studio*.

For each synonym tag (e.g., *videojs*) in the ground truth, we check if it is in a synonym group that our approach identifies, and if so, we further check if its corresponding master tag (e.g., *video.js*) is also in the synonym group. As Stack Overflow tag synonyms sometimes involve abbreviations, such as (*js*, *javascript*), we also check if a synonym tag is an abbreviation of a synonym group and if the master tag is in the corresponding synonym group.

We compare our approach with the two baselines, Wordnet and SEWordSim. WordNet [85] is a general-purpose lexical database of English created by lexicographers. WordNet groups English words into synonym sets (synsets) such as {*small*, *little*, *minor*}. For each synonym tag in the ground truth, we check if it is in the WordNet, and if so, we further check if the master tag is in the same synset as the synonym tag in the WordNet. SEWordSim [97] is a software-specific word similarity database that is extracted from Stack Overflow. For each synonym tag in ground truth, we check if it is in the SEWordSim database, and if so, we further check if the master tag is in the list of the top-20 most similar words for the synonym tag in the SEWordSim database.

²Some terms do not have abbreviations or synonyms

TABLE 6.4: The coverage of synonyms in three methods.

Method	#CoveredSynonym	#CoveredMaster	Coverage
SEthesaurus	2,316	1,439	62.1%
WordNet	725	218	30.7%
SEWordSim	941	86	9.1%

Table 6.4 summarizes the results. Overall, 2,316 (71.7%) out of 3,231 synonym tags are covered by our synonym groups, while only 725 (22.4%) and 941 (29.1%) are contained in the WordNet and the SEWordSim database. Out of the 2,316 synonym tags, 1,439 (62.1%) correct synonyms are contained in our synonym groups. This significantly outperform the accuracy of the WordNet synonyms (30.7%) and the SEWordSim synonyms (9.1%).

We further explore why our approach misses 877 (2,316-1,439) tag synonyms. First, some synonym pairs are not morphological which is beyond our scope, such as (*sky*, *flutter*) and (*wallet*, *passbook*). Second, the Stack Overflow community sometimes merge fine-grained concepts into more general ones as tag synonyms, such as (*css-reset*, *css*), (*flash-player*, *flash*) and (*worksheet*, *excel*). However, such fine-grained terms and general terms have different meanings in the discussion text, and our approach do not regard them as synonyms.

6.3.4 Human evaluation

As shown in the above evaluation, compared with several community-curated ground truth, our thesaurus contains much more software-specific terms, and a term in our thesaurus often has several abbreviations and synonyms. Therefore, our evaluation against these community-curated ground truth shows only the correctness of a subset of abbreviations and synonyms that our approach identifies, but it does not show whether many other abbreviations and synonyms that are not included in the ground truth are correct or not.

To verify the general correctness of the abbreviations and synonyms in our thesaurus, we recruit four participants for the manual evaluation including 3 final-year undergraduate students and one research assistant with master degree majoring in computer science.

They all have several-year programming experience. We split them into two groups. For each group, we randomly sample 200 abbreviation pairs and 400 synonym pairs in our thesaurus for the evaluation. Each participant independently examines the assigned samples without any discussions. They judge the correctness of abbreviations and synonyms based on their knowledge, as well as the Wikipedia and other available online information. To avoid bias, we count only pairs which are marked as correct by both participants in a group as the correct ones. In total, 400 abbreviations and 800 synonyms are manually examined.

The human evaluation confirms that 297 (74.3%) abbreviation pairs and 686 (85.8%) synonym pairs are correct. We further investigate the reasons for those incorrect pairs. Two reasons result in the wrong abbreviation pairs. First, the rules described in Section 6.2.6 could erroneously classify terms as abbreviations, such as *istream* as the abbreviation of *inputstream*, or *64-bit os* as the abbreviation of *64-bit windows*. These pairs of terms are semantically similar, but they are not abbreviations. Second, some abbreviation errors are caused by erroneous synonyms and synonym grouping. For example, *btle* is the abbreviation of *bluetooth le* (bluetooth low energy). Our approach erroneously recognizes *bluetooth le* as the synonym of *bluetooth*. Consequently, *btle* is erroneously regarded as an abbreviation of *bluetooth*. For synonyms, most errors are caused by term pairs that are both semantically and lexically similar, but are not synonyms, such as (*minsdkversion*, *maxsdkversion*), (*notification bar*, *notification tray*) and (*schema.xml*, *schema.yml*). Other synonym errors are also caused by erroneous synonyms and synonym grouping, similar to the example of the abbreviation error (*btle*, *bluetooth*).

6.4 Usefulness Evaluation

After evaluating the quality of our thesaurus, we now demonstrate the usefulness of our thesaurus for text normalization tasks.

6.4.1 Background

NLP-based techniques have been widely used to support software engineering tasks involving text data [182–184]. As abbreviations and synonyms are commonly used in software engineering text, normalizing these abbreviations and synonyms becomes one of the fundamental steps to achieve high quality text mining results [87, 88]. Abbreviations and synonyms are often referred to as inflected (or derived) words in natural language processing. The goal of text normalization is to reduce inflected (or derived) words to their root form. Techniques developed for general English text, such as stemming [169] or WordNet lemmatization [170], are commonly adopted for software engineering text. Some work proposes domain-specific techniques to normalize source code vocabulary (e.g., expanding abbreviated identifiers), but none of existing work examines the normalization of informal software engineering text on social platforms.

6.4.2 Experiment setup

6.4.2.1 Dataset

We randomly sample 100,000 questions from Stack Overflow. To further demonstrate the generality of our thesaurus, we also randomly sample 50,000 questions from CodeProject³, which is another popular Q&A web site for computer programming. We preprocess the sampled questions in the same way as described in Section 6.2.2.

6.4.2.2 Compared methods

The task is to normalize the title and content of the sampled questions. We develop a software-specific lemmatizer powered by our thesaurus for normalizing abbreviations and synonyms in informal software engineering text. We compare the performance of our lemmatizer with the two baseline methods that are commonly used for text normalization, i.e., Porter stemming [169] and WordNet-based lemmatization [170]. For

³<http://www.codeproject.com/script/Answers/List.aspx??tab=active>

FIGURE 6.6: The tags and their different forms in the question

our lemmatizer, we reduce abbreviations and synonyms to their representative terms in our thesaurus. Porter stemming reduces inflected (or derived) words to their stems by removing derivational affixes at the end of the words. WordNet-based lemmatization reduces different forms of a word to their lemma based on WordNet synset (i.e., set of synonyms created by highly trained linguists).

6.4.2.3 Ground truth and evaluation metrics

We adopt question tags as the ground truth to evaluate the effectiveness of the text normalization. Question tags can be considered as metadata of question text. We normalize question tags in the same ways as we normalize question title and content using the three compared methods. Then, we measure the effectiveness of a text normalization method by how much percentage of tags appear in question title and content before and after text normalization. We take an average of the percentage over all the sampled questions. Essentially, we investigate how much text normalization can make question texts more consistent with question metadata. Fig. 6.6 shows an example. Before text normalization, only one of the three tags (*.net*) appears in the question title and content. After normalization using our lemmatizer, all 3 tags appear in the question title and content.

FIGURE 6.7: The average percentage of tags appearing in question text by different normalization methods

6.4.3 Results

As shown in Fig. 6.7, without text normalization, on average only 55.5% and 54.0% tags appear in the title and content of the sampled Stack Overflow and CodeProject questions, respectively. This indicates that the consistency between question texts and question metadata is low. With text normalization by our lemmatizer, the percentage is boosted to 79.3% for the sampled Stack Overflow questions, and 68.7% for the sampled CodeProject questions. Although Porter stemming and WordNet-based lemmatization can also improve the consistency between question texts and question metadata, the improvement in percentage is much smaller or only marginally, compared with our lemmatizer.

The Porter stemming can only find words with derivational affixes such as (*upload*, *uploaded*) or singular and plural forms such as (*script*, *scripts*). The WordNet-based lemmatization can recognize more synonyms based on WordNet synset, such as (*os*, *operating system*). However, WordNet is a general thesaurus and lacks many software-specific terms. In contrast, our thesaurus is mined from the vast amount of software engineering text and contain a much richer set of software-specific terms and their abbreviations and synonyms. Furthermore, our thesaurus can recognize complicated synonyms, such as (*multithreading*, *multi-thread*) and (*windows*, *windwos*) that are difficult to find using Porter stemming and WordNet lemmatization. Therefore, our domain-specific thesaurus is more suitable for software-specific text normalization than general stemming methods or general English thesaurus.

6.5 Applications from Our Approach

In this section, we build three applications based on the approach mentioned above in this chapter. We not only describe how we develop these application, but also evaluate the accuracy of them compared with other methods.

6.5.1 Software-specific spelling correction

While developers write natural language documents, such as comments, documentations, blogs and Q&A posts, it is very natural that misspellings occur. In fact, a large portion of morphological synonyms that our approach identifies in Stack Overflow text are misspellings (see Table 6.5 for examples in our thesaurus). To avoid misspellings, developers can run spell checking in their editors or IDEs. Some researchers [185, 186] also use spell checkers (e.g., Aspell⁴, Hunspell⁵) to pre-processing software engineering text. However, existing spell checkers are trained for general English text, without the knowledge about software-specific terms and their common misspellings such as the examples in Table 6.5.

6.5.1.1 Data collection

Stack Overflow is the most popular programming Q&A site partially because of the community effort to maintain the quality of the question and answers in the website [125]. In Stack Overflow, even the low-quality questions and answers will be edited by the senior users to guarantee its quality for other readers. For some edition, there are also some comments to briefly describe what they change. Among these editions, 340,120 of them are about the spelling problem as the word “spell” appear in their comments. Note that this number is highly underestimated, as not every editor will mark their edition with detailed comments. So, there is an urgent need to develop a domain-specific spelling correction tool to alleviate the community efforts so that they can focus on more important editions.

⁴<http://aspell.net/>

⁵<https://hunspell.github.io/>

TABLE 6.5: Misspelling examples in our thesaurus

Term	Misspellings
ubuntu	ubunutu
jquery	jqueury, jquey
eclipse	eclispe, eclise, eclips, eclipe
android	anroid, andoid, andriod, adroid, andorid
bootstrap	bootstarp, bootstap, boostrap, bootsrap
postgresql	postgressql, postresql, posgresql, postgesql

We first get all editing history of question titles in Stack Overflow⁶. Then we compare the edition and the original post, then only the pairs which are similar enough (Jaccard similarity larger than 0.8) are left. After that, we use the differencing tool⁷ to discriminate the replacement of words between the original tile and after-edit title. We store them in the form of tuples i.e., (*original word*, *replacement word*) We set some rules (e.g, the length difference) to filter some obvious errors which are not misspellings such as pairs (*please help to understand*, *understanding*). We take the replacement pairs with frequency larger than 10 to avoid the noise. Finally, 1829 pairs of replacement pairs are extracted.

6.5.1.2 Experiment setup

We take two well-used spelling checker, Hunspell and Aspell for comparison with our SEdic. For each pair, we firstly check if the original word can be judged as misspellings by these two methods. Once the one is judged as wrong spelling, we count it as covered. Then we further check if the edited ones in ground truth is also in the suggestion list for the given word by these two methods. We calculate the accuracy for all pairs.

Our auto-correction method based on SEthesaurus is slightly different with two baselines. The SEthesaurus is stored as a thesaurus (hashmap) with the normalized form as the key, and all its synonyms and abbreviations as the value. We count the pair as covered if the original word is in our dataset. Then for the covered pairs, if the normalized form of that original word is the replacement word in ground truth, we count it as accurate.

⁶<https://archive.org/download/stackexchange/stackoverflow.com-PostHistory.7z>

⁷<https://docs.python.org/2/library/difflib.html>

TABLE 6.6: Performance comparison for spelling correction

Method	#CoveredMisspellings	Coverage	#AccurateCorrection	Accuracy
Hunspell	1525	83.4%	1057	69.3%
Aspell	1543	84.4%	956	62.0%
SEdic	1152	63.0%	982	85.2%
Hunspell + SEdic	1696	92.7%	1349	79.5%
Aspell + SEdic	1702	93.1%	1306	76.7%

6.5.1.3 Experiment results

The results can be seen in Table 6.6. We can see the coverage rate of the Hunspell and Aspell is higher than our method. That is because there are also many misspellings in Stack Overflow which is also widely appeared in daily life such as (*anther, another*), (*shure, sure*). However, our SEdic is only about the software-specific words, so our coverage rate is lower than the other two methods. For the accuracy, our method based on SEdic outperform the other two method, achieving the accuracy of 85.2%. It indicates that although the Hunspell and Aspell can identify many misspellings, they are easy to make mistakes for correction suggestions due to the unique nature of software-specific text.

To balance the coverage and accuracy, we combine the general spell checker with the specific SEdic to process software-specific text. Table 6.6 shows that both the coverage rate ($> 90\%$) and accuracy ($\sim 80\%$) are reasonably high for the real application.

6.5.2 Software-specific text wikification

In software engineering, developers often need to access online learning resources [19, 183, 187], such as Wikipedia documents, API specification, and tutorials. They frequently reference such online resources in the Stack Overflow discussions. When referencing an online resource by its URL, developers often describe the URL by some anchor text (see Figure 6.8 for an example). The URL anchor text is an important information for linking a software-specific term (e.g., an API or a library) to its learning resources.

PHP has its own function for that: `mysql_insert_id()`

```
<p>PHP has its own function for that:
<a href="http://nl.php.net/manual/en/function.mysql-insert-id.php">mysql_insert_id()</a>
</p>
```

FIGURE 6.8: An URL and its anchor text in post #5956115

TABLE 6.7: The anchor text and its frequency for the URL

URL	https://en.wikipedia.org/wiki/Single_sign-on
anchor text	single sign on (21)
text	sso (21)
(times)	single sign-on (18)
URL	https://en.wikipedia.org/wiki/System_call
anchor text	system calls (72)
text	system call (45)
(times)	syscalls (20)
	syscall (15)

In the example in Figure 6.8, the URL is for the API `mysql_insert_id` which matches the URL anchor text. However, the URL anchor texts often do not match the software-specific terms that the URLs represent in Stack Overflow text. Even worse, a URL can be referenced many times in a big corpus like Stack Overflow, but with very different anchor texts. Table 6.7 shows two examples. When referencing the Wikipedia document about “single sign-on”, people use anchor texts like *single sign on* (21 times), *sso* (21 times), and *single sign-on* (18 times). These anchor texts can be regarded as abbreviations and synonyms. The presence of such abbreviations and synonyms makes it difficult to link a software-specific term and its learning resources.

In this section, we show how our abbreviation and synonym thesaurus can help normalize URL anchor texts so that the crowd-created descriptions of an URL can better match the “standard” description of the URL. In particular, we extract 7,108 Wikipedia URLs from Stack Overflow text which are referenced at least twice and have at least two different anchor texts for this experiment. We use the wikipedia URLs because the URLs define the standard description that can be easily extracted from the URLs. For example, from the URL https://en.wikipedia.org/wiki/System_call, we can extract the last portion of the URL *system call* as the standard description of the URL.

Using our lemmatizer, we replace anchor texts of an URL with their representative terms in our abbreviation and synonym thesaurus, and then we rerank the representative terms by the sum frequencies of being-replaced anchor texts. For example, for the “single

FIGURE 6.9: The Precision@k of the standard URL description by different normalization methods

sign-on” Wikipedia URL, all the three anchor texts will be replaced with the representative term *single sign-on*, and the frequency of this representative term is the sum of the frequency of the three anchor text being replaced. Again, we compare the performance of our lemmatizer against Porter stemming [169] and WordNet-based lemmatization [170]. We perform the replacement and rerank process using Porter stemming and WordNet-based lemmatization in the same way as using our lemmatizer.

To measure the effectiveness of an anchor text normalization method, we adopt Precision@k ($k=1, 2$ in this evaluation⁸) as the metric i.e., whether the standard description of an URL appears before the rank k in the normalized anchor text list. As seen in Figure 6.9, the Precision@1 after anchor text normalization using our lemmatizer is 0.88, which is a significant boost compared with 0.72 without anchor text normalization, 0.75 with Porter stemming and 0.73 with WordNet-based lemmatization. Similar observation can be made for Precision@2. This result demonstrates that our thesaurus can potentially improve traceability recovery between a software-specific term and its learning resources [187–189].

⁸We use small k values because the number of anchor texts of an URL vary greatly, but all the URLs in our experiment have at least two anchor texts.

6.5.3 Identify tag synonyms on Stack Overflow

There are tens of thousands of tags in Stack Overflow proposed by different users in the community. Due to the diversity of the human language, it is very likely that same-meaning tags with slightly different forms co-exist in the site (e.g., *pdfjs*⁹ *pdf.js*¹⁰). Such synonym tags may cause the confusion among users and make it difficult to retrieve posts by tags.

Although the community has proposed a list of synonym tags¹¹, there are still many more synonyms which are not discovered. In this section, we directly apply our approach in finding tag synonyms to complement existing synonym list curated by the community manually. Note as we need to mine synonym tags, so we change the dataset from text to all tags attached to questions in Stack Overflow.

In Stack Overflow, there are 9,970,064 questions and each sentence must be attached with up to 5 tags. Therefore, we regard tags for each question as a sentence, and feed them into word embedding model (more details can be referred by our previous word [100]). For each tag, given its semantic words from word embedding, we further exploit the rule-based methods proposed above in this chapter to identify its abbreviations and synonyms. After manual checking, we extract 916 pairs of synonyms and abbreviations.

We paste all accurate pairs into a post¹² in Meta Stack Overflow which is a site where user can discuss the workings and policies of Stack Overflow rather than discussing programming itself. So far, it has received 65 votes, 9 favorite question marks and some appreciating comments. We believe that this list contribute to this community.

⁹<http://stackoverflow.com/tags/pdfjs/info>

¹⁰<http://stackoverflow.com/tags/pdf.js/info>

¹¹<http://stackoverflow.com/tags/synonyms>

¹²<http://meta.stackoverflow.com/questions/342097/a-list-of-tag-synonyms-not-proposed-in-stack-overflow>

TABLE 6.8: Exmaples of semantically related techniques

Term	Semantically related terms
java	scala, groovy, c++, clojure, c#, delphi, python
netbeans	eclipse, intellij, pydev, android stuio, aptana
beautifulsoup	lxml, nokogiri, html agility pack, jsoup, simplexml
codeigniter	cakephp, yii, zend, symfony, django, joomla, laravel
binary search	linear search, bubble sort, radix sort, quicksort

6.5.4 Building software dictionary

Although our thesaurus is useful for many applications mentioned above, it can also be further enriched to become a software-specific dictionary. In this section, we incorporate different information sources mined from Stack Overflow to build a practical software-related dictionary.

All software-specific terms mined in our approach can be regarded as an entry in our dictionary. For each entry, there are four different kinds of information:

- its abbreviations and synonyms;
- the definition from Wikipedia or TagWiki in Stack Overflow;
- the semantic related terms to it excluding the abbreviation and synonyms;
- its frequent links in Stack Overflow excluding the Wikipedia or Stack Overflow links.

The abbreviations and synonyms of each term come from our SEthesaurus. We find its Wikipedia or Stack Overflow links according to the anchor text of the link which is similar to the procedures in Section 6.5.2. The semantic related terms are mined from Section 6.2.5. After extracting the synonyms and abbreviations, the rest is a list of terms which are highly correlated with the given term. Table 6.8 shows some examples, and they can be exploited in recommendation systems for software engineering tasks. For example, we could use this knowledge to recommend similar techniques or analogical techniques across programming languages [100]. We may also exploit these semantically related terms for query expansion and reformulation [190, 191]. The frequent

FIGURE 6.10: The screen shot of our website

FIGURE 6.11: The screen shot of the Firefox extension in use

links are extracted in the same way as the obtain of the Wikipedia or Stack Overflow links.

We build a prototype demo website¹³ for the community to access to our dictionary (Figure 6.10). In the website, users can search for software-specific terms and find their abbreviations and synonyms. As our dictionary can also be applied to other related tasks, we also release the API (similar to WordNet API) for developers to access to the information in the dictionary. As an examples, we develop a Firefox plug-in¹⁴ based on our dictionary which can assist user in understanding software-related articles by showing the explanation for their interested software-specific terms. The screen shot can be seen in Figure 6.11.

¹³<https://se-dictionary.appspot.com/>

¹⁴<https://github.com/ccywch/se-dict-browser-extension>

6.6 Conclusions

In this chapter, we present an automatic approach for mining a thesaurus of software-specific terms and commonly-used morphological forms from informal software engineering discussions. It can be used to expand the entities in our knowledge graph so that the knowledge graph can be more robust to different queries written by different developers.

Our evaluation shows that our thesaurus covers a large set of software-specific terms, abbreviations and synonyms with high accuracy. In a text normalization task, we demonstrate that our thesaurus can significantly improve the consistency between question text and question metadata, compared with general stemming and lemmatization methods.

Chapter 7

Conclusions and Future Research

In this chapter, we summarize the research work that we have conducted in the thesis, and then discuss our future research direction and work.

7.1 Summary of Completed Work

During the last decade, Stack Overflow has accumulated millions of questions and answers which contains a great amount of software development experience. It has emerged as an invaluable resource for developers around the world. But its large-scale and unstructured characteristics bring a great challenge to efficient retrieval. My thesis try to solve such problems from the following two aspects.

First, to make content more easily to be searched by users, I develop an edit-assistance tool for identifying quality issues in Stack Overflow posts and recommending sentence edits for correction. It corrects abbreviation, misspellings, especially errors of software-specific terms so that users will not miss important information in the site.

As the method is developed by analyzing the data from Stack Overflow, it can be incorporated seamlessly into Stack Overflow, or other sites within Stack Exchange. But note that without a large-scale, live deployment, I can't guarantee that the applications based on my proposed algorithms can make Stack Overflow better. Instead, what I proposed

may bring unintended consequences. For example, although automating minor edits can improve the quality of content in Stack Overflow which is better for searching, it may also close peripheral contributors a way to enter the community. In Stack Overflow, new users can propose to edit the post with earning credit scores, and that can be regarded as gateway tasks for new community members. Introducing technology to replace these tasks might harm the community since it will have to find alternative approaches to familiarizing new members with community norms and provide them with things to do. So, applying our technology into Stack Overflow not only requires the consideration of technical details, but also the society.

Second, to support implicit knowledge search and direct answers to users' query, I build a software-specific knowledge graph by mining tags in Stack Overflow. Then I enrich the naive knowledge graph with more fine-grained relationships i.e., analogical libraries across different programming languages, by incorporating the tag semantics from word embedding, relational information, and categorical information. To further make the knowledge-graph based entity search robust, we infer morphological forms of software-specific terms by combining distributed word semantics, domain-specific lexical rules and transformations, and graph analysis of morphological relations.

Based on the knowledge graph, I have built several websites to support developers' knowledge search. And these websites do attract many users all around the world. They demonstrated the usefulness of my knowledge graph in software-engineering domain. Google has already built a large-scale knowledge graph¹ as the backend for its searching. But it is built for general purpose without having any emphasis on developers, and I do believe that combining my knowledge graph can further enrich theirs for better serving developers' queries. In addition, one of my plugins seen in Figure 4.4 provides another way to complement Google Search with mined explicit knowledge in my knowledge graph. The tool can render direct high-level answers to some queries of developers, and developers can also look for more detailed information within the Google search results at the same page.

¹<https://www.google.com/intl/bn/insidesearch/features/search/knowledge.html>

FIGURE 7.1: The overview of finished works and future works

7.2 Future Work

The goal of my future research is to provide more tools for further enhancing the developers' productivity from different aspects. One aspect of this agenda is to extend my current research horizontally into more applications such as software security, chatbot. But note that there are many parameters within our models, and these parameters are tailored specifically for Stack Overflow data. For some parameters which may significantly influence the model performance such as the support value and confidence value in Chapter 4, they are selected by trying different settings with formal experiments. However, most of them are not so sensitive, hence determined heuristically by small-scale pilot studies. Therefore, when extending our methodology into other datasets, the parameters may need to be adjusted accordingly.

Another research direction is to broaden my research scope vertically, both to lower-level software artifacts such as source code and binary code, or higher level like UI design. The overview of the current works and future works can be seen in Figure 7.1. In the future, we are planning to conduct the following works:

- **Knowledge graph for software security.** Software vulnerabilities or bugs consume significant resources in the lifecycle of a software product development. Current practices focus mainly on reacting to vulnerabilities when they occur. But we believe that the community needs more research on proactive methods that can increase the developers' awareness of potential vulnerability and thus prevent them from occurring in the first place. We think an API-caveats or software vulnerabilities knowledge graph can serve this purpose. We will distill such domain-specific knowledge graph from three general categories of information sources: common knowledge (e.g., Wikipedia pages on encryption, Common Weakness Enumeration database), producer knowledge (e.g., API documents), and consumer knowledge (e.g., technical forum discussions, Common Vulnerabilities and Exposures database). By turning mostly textual information into a structured knowledge base, software developers can easily understand the landscape of common vulnerabilities and the relationships among them so that they can avoid potential bugs or vulnerabilities in their programs.
- **Chatbot system for software-specific question answering.** When software developers encounter problems, they are very likely to search the web for possible technical solutions. Although the search engine may return many related links, the overwhelming information makes it difficult for developers to digest, or find the most needed answer to their questions. So, inspired by the current progress of the chatbot, I am also trying to build a software-specific chatbot to render direct answers to developers' questions. Our data mainly comes from two important information resources. One is the Q&A website like Stack Overflow with not only questions and answers, but also high-voted comments. The other resource is the historical discussion logs of previous projects in their specific online chatroom (e.g., the Internet Relay Chat of different open-source projects such as Eclipse, Ubuntu, OpenStack, etc). Based on these two kinds of data, we are trying to develop a neural-network model to learn the direct answers from Q&A data, and also the communication style from the discussion log. With this robot, developers can get a direct answer to their problems via chatting with the bot, and they only need to resort to the search engine if they are not satisfied or want more details.

- **Deep learning for mining binary code.** To make the software bug-free and secure, extensive and large-scale testing is necessary. But white box testing may not be applicable as the source code for the target software is very likely to be inaccessible. So, blackbox or graybox testing are often needed. i.e., testing on the binary code or with partial source code. However, the large amount of binary code is much more lengthy, and difficult to understand than well-documented source code. A tool to help security scientists to locate security-sensitive components is needed so that they can focus on such important parts which are likely to invite vulnerabilities. I am planning to develop a deep learning based classification model to identify the functionality of binary code segmentations, especially those highly related to security issues such as authentication, access control, and interprocess communication.
- **UI design generation.** Good GUI design is difficult and time-consuming, even for professional designers. A nice GUI design involves many golden rules such as fluent interactivity, universal usability, clear readability, aesthetic appearance, consistent styles, and etc [192]. However, such heavy works await just a few designers within the company. For example, the ratio of developers to user interface designers at Microsoft was 50:1, and this ratio was better than any other company out there [193]. So, some assistance tools are terribly needed to help alleviate the heavy burden on designers. We have collected many UI screenshots [194] from the top commercial Apps in Google Play i.e., we can assume these UI design as the good designs. Then we are trying to implement a deep learning model to generate some UI designs according to designers' certain requirements. With the generated UI design, the designers can select candidates what they like most and fine-tune the UI design which will definitely reduce their workload.

Bibliography

- [1] Chunyang Chen, Zhenchang Xing, and Yang Liu. By the community & for the community: A deep learning approach to assist collaborative editing in q&a sites. In *Proceedings of the 21st ACM Conference on Computer-Supported Cooperative Work and Social Computing*, pages 32:1–32:21. ACM, 2018.
- [2] Chunyang Chen and Zhenchang Xing. Mining technology landscape from stack overflow. In *Proceedings of the 10th ACM/IEEE International Symposium on Empirical Software Engineering and Measurement*, page 14. ACM, 2016.
- [3] Chunyang Chen, Zhenchang Xing, and Lei Han. Techland: Assisting technology landscape inquiries with insights from stack overflow. In *Software Maintenance and Evolution (ICSME), 2016 IEEE International Conference on*, pages 356–366. IEEE, 2016.
- [4] Chunyang Chen, Sa Gao, and Zhenchang Xing. Mining analogical libraries in q&a discussions—incorporating relational and categorical knowledge into word embedding. In *Software Analysis, Evolution, and Reengineering (SANER), 2016 IEEE 23rd International Conference on*, volume 1, pages 338–348. IEEE, 2016.
- [5] Chunyang Chen and Zhenchang Xing. Similartech: Automatically recommend analogical libraries across different programming languages. In *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering*, pages 834–839. ACM, 2016.

- [6] Yi Huang, Chunyang Chen, Zhenchang Xing, Tian Lin, and Yang Liu. Tell them apart: Distilling technology differences from crowd-scale comparison discussions. In *Proceedings of the 33st IEEE/ACM International Conference on Automated Software Engineering*. IEEE Press, 2018.
- [7] Chunyang Chen, Zhenchang Xing, and Ximing Wang. Unsupervised software-specific morphological forms inference from informal discussions. In *Proceedings of the 39th International Conference on Software Engineering*, pages 450–461. IEEE Press, 2017.
- [8] 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*, 19(3):619–654, 2014.
- [9] 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.
- [10] 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.
- [11] 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.
- [12] Aditya Pal, Shuo Chang, and Joseph A Konstan. Evolution of experts in question answering communities. In *ICWSM*, 2012.
- [13] 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.

- [14] Latifa Guerrouj, Shams Azad, and Peter 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. IEEE, 2015.
- [15] Santanu Paul and Atul Prakash. A framework for source code search using program patterns. *Software Engineering, IEEE Transactions on*, 20(6):463–475, 1994.
- [16] Otávio Augusto Lazzarini Lemos, Sushil Krishna Bajracharya, Joel Ossher, Ricardo Santos Morla, Paulo Cesar Masiero, Pierre Baldi, and Cristina Videira Lopes. Codegenie: using test-cases to search and reuse source code. In *Proceedings of the twenty-second IEEE/ACM international conference on Automated software engineering*, pages 525–526. ACM, 2007.
- [17] Steven P Reiss. Semantics-based code search. In *Proceedings of the 31st International Conference on Software Engineering*, pages 243–253. IEEE Computer Society, 2009.
- [18] Shaowei Wang, David Lo, and Lingxiao Jiang. Active code search: incorporating user feedback to improve code search relevance. In *Proceedings of the 29th ACM/IEEE international conference on Automated software engineering*, pages 677–682. ACM, 2014.
- [19] Siddharth Subramanian, Laura Inozemtseva, and Reid Holmes. Live api documentation. In *ICSE*, pages 643–652. ACM, 2014.
- [20] Martin P Robillard and Yam B Chhetri. Recommending reference api documentation. *Empirical Software Engineering*, 20(6):1558–1586, 2015.
- [21] Ripon K Saha, Matthew Lease, Sarfraz Khurshid, and Dewayne E Perry. Improving bug localization using structured information retrieval. In *Automated Software Engineering (ASE), 2013 IEEE/ACM 28th International Conference on*, pages 345–355. IEEE, 2013.

- [22] 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.
- [23] Bogdan Dit, Meghan Revelle, and Denys Poshyvanyk. Integrating information retrieval, execution and link analysis algorithms to improve feature location in software. *Empirical Software Engineering*, 18(2):277–309, 2013.
- [24] Daniel Hasan Dalip, Marcos André Gonçalves, Marco Cristo, and Pavel Calado. Exploiting user feedback to learn to rank answers in q&a forums: a case study with stack overflow. In *Proceedings of the 36th international ACM SIGIR conference on research and development in information retrieval*, pages 543–552. ACM, 2013.
- [25] Yanzhen Zou, Ting Ye, Yangyang Lu, John Mylopoulos, and Lu Zhang. Learning to rank for question-oriented software text retrieval (t). In *Automated Software Engineering (ASE), 2015 30th IEEE/ACM International Conference on*, pages 1–11. IEEE, 2015.
- [26] Xin Xia, David Lo, Xingen Wang, Chenyi Zhang, and Xinyu Wang. Cross-language bug localization. In *Proceedings of the 22nd International Conference on Program Comprehension*, pages 275–278. ACM, 2014.
- [27] Jane Huffman Hayes, Hakim Sultanov, Wei-Keat Kong, and Wenbin Li. Software verification and validation research laboratory (svvrl) of the university of kentucky: traceability challenge 2011: language translation. In *Proceedings of the 6th International Workshop on Traceability in Emerging Forms of Software Engineering*, pages 50–53. ACM, 2011.
- [28] Bowen Xu, Zhenchang Xing, Xin Xia, David Lo, Qingye Wang, and Shanping Li. Domain-specific cross-language relevant question retrieval. In *Proceedings of the 13th International Conference on Mining Software Repositories*, page (To appear). IEEE, 2016.

- [29] Marcin Junczys-Dowmunt and Roman Grundkiewicz. Phrase-based machine translation is state-of-the-art for automatic grammatical error correction. *arXiv preprint arXiv:1605.06353*, 2016.
- [30] Tomoya Mizumoto and Yuji Matsumoto. Discriminative reranking for grammatical error correction with statistical machine translation. In *Proceedings of NAACL-HLT*, pages 1133–1138, 2016.
- [31] Zheng Yuan, Ted Briscoe, and Mariano Felice. Candidate re-ranking for smt-based grammatical error correction. In *Proceedings of the 11th Workshop on Innovative Use of NLP for Building Educational Applications*, pages 256–266, 2016.
- [32] Philipp Koehn. *Statistical machine translation*. Cambridge University Press, 2009.
- [33] Piotr Mirowski and Andreas Vlachos. Dependency recurrent neural language models for sentence completion. *arXiv preprint arXiv:1507.01193*, 2015.
- [34] Yonghui Wu, Mike Schuster, Zhifeng Chen, Quoc V Le, Mohammad Norouzi, Wolfgang Macherey, Maxim Krikun, Yuan Cao, Qin Gao, Klaus Macherey, et al. Google’s neural machine translation system: Bridging the gap between human and machine translation. *arXiv preprint arXiv:1609.08144*, 2016.
- [35] Kyunghyun Cho, Bart Van Merriënboer, Caglar Gulcehre, Dzmitry Bahdanau, Fethi Bougares, Holger Schwenk, and Yoshua Bengio. Learning phrase representations using rnn encoder-decoder for statistical machine translation. *arXiv preprint arXiv:1406.1078*, 2014.
- [36] Robert Dale and Adam Kilgarriff. Helping our own: The hoo 2011 pilot shared task. In *Proceedings of the 13th European Workshop on Natural Language Generation*, pages 242–249. Association for Computational Linguistics, 2011.
- [37] Hwee Tou Ng, Siew Mei Wu, Ted Briscoe, Christian Hadiwinoto, Raymond Hendy Susanto, and Christopher Bryant. The conll-2014 shared task on grammatical error correction. In *CoNLL Shared Task*, pages 1–14, 2014.

- [38] Zhuoran Liu and Yang Liu. Exploiting unlabeled data for neural grammatical error detection. *arXiv preprint arXiv:1611.08987*, 2016.
- [39] Mariano Felice. Artificial error generation for translation-based grammatical error correction. Technical report, University of Cambridge, Computer Laboratory, 2016.
- [40] Ziang Xie, Anand Avati, Naveen Arivazhagan, Dan Jurafsky, and Andrew Y Ng. Neural language correction with character-based attention. *arXiv preprint arXiv:1603.09727*, 2016.
- [41] Allen Schmaltz, Yoon Kim, Alexander M Rush, and Stuart M Shieber. Sentence-level grammatical error identification as sequence-to-sequence correction. *arXiv preprint arXiv:1604.04677*, 2016.
- [42] Shamil Chollampatt, Kaveh Taghipour, and Hwee Tou Ng. Neural network translation models for grammatical error correction. In *Proceedings of the Twenty-Fifth International Joint Conference on Artificial Intelligence, IJCAI 2016, New York, NY, USA, 9-15 July 2016*, pages 2768–2774, 2016.
- [43] Zheng Yuan and Ted Briscoe. Grammatical error correction using neural machine translation. In *Proceedings of NAACL-HLT*, pages 380–386, 2016.
- [44] Edgar Meij, Krisztian Balog, and Daan Odijk. Entity linking and retrieval for semantic search. In *WSDM*, pages 683–684, 2014.
- [45] Mihai Surdeanu, Julie Tibshirani, Ramesh Nallapati, and Christopher D Manning. Multi-instance multi-label learning for relation extraction. In *Proceedings of the 2012 Joint Conference on EMNLP-CoNLL*, pages 455–465. Association for Computational Linguistics, 2012.
- [46] Alexander Yates, Michael Cafarella, Michele Banko, Oren Etzioni, Matthew Broadhead, and Stephen Soderland. Textrunner: open information extraction on the web. In *Proceedings of Human Language Technologies: The Annual Conference of the North American Chapter of the ACL: Demonstrations*, pages 25–26. Association for Computational Linguistics, 2007.

- [47] Jun Zhu, Zaiqing Nie, Xiaojiang Liu, Bo Zhang, and Ji-Rong Wen. Statsnowball: a statistical approach to extracting entity relationships. In *WWW*, pages 101–110. ACM, 2009.
- [48] Nikos Sarkas, Stelios Paparizos, and Panayiotis Tsaparas. Structured annotations of web queries. In *ACM SIGMOD*, pages 771–782. ACM, 2010.
- [49] Surender Reddy Yerva, Zoltan Miklos, Flavia Grosan, Alexandru Tandrau, and Karl Aberer. Tweetspector: Entity-based retrieval of tweets. In *SIGIR*, pages 1016–1016. ACM, 2012.
- [50] Yegin Genc. Exploratory search with semantic transformations using collaborative knowledge bases. In *WSDM*, pages 661–666. ACM, 2014.
- [51] Ilaria Bordino, Yelena Mejova, and Mounia Lalmas. Penguins in sweaters, or serendipitous entity search on user-generated content. In *22nd ACM CIKM*, pages 109–118. ACM, 2013.
- [52] Margaret-Anne Storey, Li-Te Cheng, Ian Bull, and Peter Rigby. Shared waypoints and social tagging to support collaboration in software development. In *Proceedings of the 20th anniversary conference on Computer supported cooperative work*, pages 195–198. ACM, 2006.
- [53] Christoph Treude and M Storey. How tagging helps bridge the gap between social and technical aspects in software development. In *ICSE*, pages 12–22. IEEE, 2009.
- [54] Seyed Mehdi Nasehi, Jonathan Sillito, Frank Maurer, and Chris Burns. What makes a good code example?: A study of programming q&a in stackoverflow. In *28th ICSM*, pages 25–34. IEEE, 2012.
- [55] Harry Halpin, Valentin Robu, and Hana Shepherd. The complex dynamics of collaborative tagging. In *WWW*, pages 211–220. ACM, 2007.
- [56] George Macgregor and Emma McCulloch. Collaborative tagging as a knowledge organisation and resource discovery tool. *Library review*, 55(5):291–300, 2006.

- [57] Valentin Robu, Harry Halpin, and Hana Shepherd. Emergence of consensus and shared vocabularies in collaborative tagging systems. *ACM Transactions on the Web (TWEB)*, 3(4):14, 2009.
- [58] Denis Helic, Markus Strohmaier, Christoph Trattner, Markus Muhr, and Kristina Lerman. Pragmatic evaluation of folksonomies. In *WWW*, pages 417–426. ACM, 2011.
- [59] Edwin Simpson. Clustering tags in enterprise and web folksonomies. In *ICWSM*, 2008.
- [60] Shaowei Wang, David Lo, and Lingxiao Jiang. Inferring semantically related software terms and their taxonomy by leveraging collaborative tagging. In *ICSM*, pages 604–607. IEEE, 2012.
- [61] Patrick Schmitz. Inducing ontology from flickr tags. In *Collaborative Web Tagging Workshop at WWW*, volume 50, 2006.
- [62] Mark Sanderson and Bruce Croft. Deriving concept hierarchies from text. In *SIGIR*, pages 206–213. ACM, 1999.
- [63] Yuan Tian, David Lo, and Julia Lawall. Automated construction of a software-specific word similarity database. In *CSMR-WCRE*, pages 44–53. IEEE, 2014.
- [64] Jinqiu Yang and Lin Tan. Inferring semantically related words from software context. In *MSR*, pages 161–170. IEEE, 2012.
- [65] Jeanne Ferrante, Karl J Ottenstein, and Joe D Warren. The program dependence graph and its use in optimization. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 9(3):319–349, 1987.
- [66] James Rumbaugh, Ivar Jacobson, and Grady Booch. *Unified Modeling Language Reference Manual, The*. Pearson Higher Education, 2004.
- [67] Tung Thanh Nguyen, Anh Tuan Nguyen, Hoan Anh Nguyen, and Tien N Nguyen. A statistical semantic language model for source code. In *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering*, pages 532–542. ACM, 2013.

- [68] Cédric Teyton, Jean-Rémy Falleri, and Xavier Blanc. Automatic discovery of function mappings between similar libraries. In *Reverse Engineering (WCRE), 2013 20th Working Conference on*, pages 192–201. IEEE, 2013.
- [69] Hao Zhong, Suresh Thummalapenta, Tao Xie, Lu Zhang, and Qing Wang. Mining api mapping for language migration. In *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering-Volume 1*, pages 195–204. ACM, 2010.
- [70] Anh Tuan Nguyen, Hoan Anh Nguyen, Tung Thanh Nguyen, and Tien N Nguyen. Statistical learning approach for mining api usage mappings for code migration. In *Proceedings of the 29th ACM/IEEE international conference on Automated software engineering*, pages 457–468. ACM, 2014.
- [71] Ferdian Thung, Daniel Lo, and Julia Lawall. Automated library recommendation. In *Reverse Engineering (WCRE), 2013 20th Working Conference on*, pages 182–191. IEEE, 2013.
- [72] Cédric Teyton, Jean-Rémy Falleri, and Xavier Blanc. Mining library migration graphs. In *Reverse Engineering (WCRE), 2012 19th Working Conference on*, pages 289–298. IEEE, 2012.
- [73] Cédric Teyton, Jean-Rémy Falleri, Marc Palyart, and Xavier Blanc. A study of library migrations in java. *Journal of Software: Evolution and Process*, 26(11): 1030–1052, 2014.
- [74] 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.
- [75] Peter D Turney. Similarity of semantic relations. *Computational Linguistics*, 32(3):379–416, 2006.
- [76] Chang Xu, Yalong Bai, Jiang Bian, Bin Gao, Gang Wang, Xiaoguang Liu, and Tie-Yan Liu. Rc-net: A general framework for incorporating knowledge into word representations. In *Proceedings of the 23rd ACM International Conference*

- on Conference on Information and Knowledge Management*, pages 1219–1228. ACM, 2014.
- [77] 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.
- [78] Phong Minh Vu, Tam The Nguyen, Hung Viet Pham, and Tung Thanh Nguyen. Mining user opinions in mobile app reviews: A keyword-based approach (t). In *Automated Software Engineering (ASE), 2015 30th IEEE/ACM International Conference on*, pages 749–759. IEEE, 2015.
- [79] Phong Minh Vu, Hung Viet Pham, Tung Thanh Nguyen, et al. Phrase-based extraction of user opinions in mobile app reviews. In *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering*, pages 726–731. ACM, 2016.
- [80] Xin Ye, Hui Shen, Xiao Ma, Razvan Bunescu, and Chang Liu. From word embeddings to document similarities for improved information retrieval in software engineering. In *Proceedings of the 38th international conference on software engineering*, pages 404–415. ACM, 2016.
- [81] Thanh Van Nguyen, Anh Tuan Nguyen, and Tien N Nguyen. Characterizing api elements in software documentation with vector representation. In *Proceedings of the 38th International Conference on Software Engineering Companion*, pages 749–751. ACM, 2016.
- [82] Trong Duc Nguyen, Anh Tuan Nguyen, and Tien N Nguyen. Mapping api elements for code migration with vector representations. In *Proceedings of the 38th International Conference on Software Engineering Companion*, pages 756–758. ACM, 2016.

- [83] Bowen Xu, Deheng Ye, Zhenchang Xing, Xin Xia, Guibin Chen, and Shanping Li. Predicting semantically linkable knowledge in developer online forums via convolutional neural network. In *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering*, pages 51–62. ACM, 2016.
- [84] Guibin Chen, Chunyang Chen, Zhenchang Xing, and Bowen Xu. Learning a dual-language vector space for domain-specific cross-lingual question retrieval. In *Automated Software Engineering (ASE), 2016 31st IEEE/ACM International Conference on*, pages 744–755. IEEE, 2016.
- [85] George A Miller. Wordnet: a lexical database for english. *Communications of the ACM*, 38(11):39–41, 1995.
- [86] Hongfang Liu, Zhang-Zhi Hu, Jian Zhang, and Cathy Wu. Biothesaurus: a web-based thesaurus of protein and gene names. *Bioinformatics*, 22(1):103–105, 2006.
- [87] Dawn Lawrie, Henry Feild, and David Binkley. Extracting meaning from abbreviated identifiers. In *Seventh IEEE International Working Conference on Source Code Analysis and Manipulation (SCAM 2007)*, pages 213–222. IEEE, 2007.
- [88] Dawn Lawrie, Dave Binkley, and Christopher Morrell. Normalizing source code vocabulary. In *2010 17th Working Conference on Reverse Engineering*, pages 3–12. IEEE, 2010.
- [89] Bogdan Dit, Latifa Guerrouj, Denys Poshyvanyk, and Giuliano Antoniol. Can better identifier splitting techniques help feature location? In *Program Comprehension (ICPC), 2011 IEEE 19th International Conference on*, pages 11–20. IEEE, 2011.
- [90] Dawn Lawrie and Dave Binkley. Expanding identifiers to normalize source code vocabulary. In *Software Maintenance (ICSM), 2011 27th IEEE International Conference on*, pages 113–122. IEEE, 2011.

- [91] Latifa Guerrouj, Philippe Galinier, Yann-Gaël Guéhéneuc, Giuliano Antoniol, and Massimiliano Di Penta. Tris: A fast and accurate identifiers splitting and expansion algorithm. In *2012 19th Working Conference on Reverse Engineering*, pages 103–112. IEEE, 2012.
- [92] David Shepherd, Zachary P Fry, Emily Hill, Lori Pollock, and K Vijay-Shanker. Using natural language program analysis to locate and understand action-oriented concerns. In *Proceedings of the 6th international conference on Aspect-oriented software development*, pages 212–224. ACM, 2007.
- [93] Giriprasad Sridhara, Emily Hill, Lori Pollock, and K Vijay-Shanker. Identifying word relations in software: A comparative study of semantic similarity tools. In *Program Comprehension, 2008. ICPC 2008. The 16th IEEE International Conference on*, pages 123–132. IEEE, 2008.
- [94] 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.
- [95] Jinqiu Yang and Lin Tan. Swordnet: Inferring semantically related words from software context. *Empirical Software Engineering*, 19(6):1856–1886, 2014.
- [96] David Shepherd, Lori Pollock, and K Vijay-Shanker. Towards supporting on-demand virtual remodularization using program graphs. In *Proceedings of the 5th international conference on Aspect-oriented software development*, pages 3–14. ACM, 2006.
- [97] Yuan Tian, David Lo, and Julia Lawall. Sewordsim: Software-specific word similarity database. In *Companion Proceedings of the 36th International Conference on Software Engineering*, pages 568–571. ACM, 2014.
- [98] Chen Guibin, Chunyang Chen, Zhenchang Xing, and Xu Bowen. Learning a dual-language vector space for domain-specific cross-lingual question retrieval. In *31st IEEE/ACM International Conference on Automated Software Engineering (ASE). IEEE/ACM*, 2016.

- [99] Deheng Ye, Zhenchang Xing, Chee Yong Foo, Jing Li, and Nachiket Kapre. Learning to extract api mentions from informal natural language discussions. In *2016 IEEE 23rd International Conference on Software Analysis, Evolution, and Reengineering (SANER)*, volume 1, pages 90–101. IEEE, 2016.
- [100] Chunyang Chen, Sa Gao, and Zhenchang Xing. Mining analogical libraries in q&a discussions -incorporating relational and categorical knowledge into word embedding. In *The 23rd SANER*, pages 338–348. IEEE, 2016.
- [101] Bowen Xu, Deheng Ye, Zhenchang Xing, Xin Xia, Chen Guibin, and Li Shanping. Predicting semantically linkable knowledge in developer online forums via convolutional neural network. In *31st IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE/ACM, 2016.
- [102] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. Imagenet classification with deep convolutional neural networks. In *Advances in neural information processing systems*, pages 1097–1105, 2012.
- [103] Siwei Lai, Liheng Xu, Kang Liu, and Jun Zhao. Recurrent convolutional neural networks for text classification. In *AAAI*, volume 333, pages 2267–2273, 2015.
- [104] Geoffrey Hinton, Li Deng, Dong Yu, George E Dahl, Abdel-rahman Mohamed, Navdeep Jaitly, Andrew Senior, Vincent Vanhoucke, Patrick Nguyen, Tara N Sainath, et al. Deep neural networks for acoustic modeling in speech recognition: The shared views of four research groups. *IEEE Signal Processing Magazine*, 29(6):82–97, 2012.
- [105] Trong Duc Nguyen, Anh Tuan Nguyen, Hung Dang Phan, and Tien N Nguyen. Exploring api embedding for api usages and applications. In *Software Engineering (ICSE), 2017 IEEE/ACM 39th International Conference on*, pages 438–449. IEEE, 2017.
- [106] Xiaodong Gu, Hongyu Zhang, Dongmei Zhang, and Sunghun Kim. Deep api learning. In *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, pages 631–642. ACM, 2016.

- [107] Matej Balog, Alexander L Gaunt, Marc Brockschmidt, Sebastian Nowozin, and Daniel Tarlow. Deepcoder: Learning to write programs. *arXiv preprint arXiv:1611.01989*, 2016.
- [108] Carol V Alexandru. Guided code synthesis using deep neural networks. In *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, pages 1068–1070. ACM, 2016.
- [109] Akshay Sethi, Anush Sankaran, Naveen Panwar, Shreya Khare, and Senthil Mani. Dlpaper2code: Auto-generation of code from deep learning research papers. *arXiv preprint arXiv:1711.03543*, 2017.
- [110] Chunyang Chen, Ting Su, Guozhu Meng, Zhenchang Xing, and Yang Liu. From ui design image to gui skeleton: A neural machine translator to bootstrap mobile gui implementation. In *The 40th International Conference on Software Engineering, Gothenburg, Sweden*. ACM, 2018.
- [111] Tony Beltramelli. pix2code: Generating code from a graphical user interface screenshot. *arXiv preprint arXiv:1705.07962*, 2017.
- [112] Veselin Raychev, Martin Vechev, and Eran Yahav. Code completion with statistical language models. In *Acm Sigplan Notices*, volume 49, pages 419–428. ACM, 2014.
- [113] Martin White, Christopher Vendome, Mario Linares-Vásquez, and Denys Poshyvanyk. Toward deep learning software repositories. In *Mining Software Repositories (MSR), 2015 IEEE/ACM 12th Working Conference on*, pages 334–345. IEEE, 2015.
- [114] Hao Peng, Lili Mou, Ge Li, Yuxuan Liu, Lu Zhang, and Zhi Jin. Building program vector representations for deep learning. In *International Conference on Knowledge Science, Engineering and Management*, pages 547–553. Springer, 2015.

- [115] Lili Mou, Ge Li, Lu Zhang, Tao Wang, and Zhi Jin. Convolutional neural networks over tree structures for programming language processing. In *AAAI*, volume 2, page 4, 2016.
- [116] An Ngoc Lam, Anh Tuan Nguyen, Hoan Anh Nguyen, and Tien N Nguyen. Combining deep learning with information retrieval to localize buggy files for bug reports (n). In *Automated Software Engineering (ASE), 2015 30th IEEE/ACM International Conference on*, pages 476–481. IEEE, 2015.
- [117] Jin Guo, Jinghui Cheng, and Jane Cleland-Huang. Semantically enhanced software traceability using deep learning techniques. In *Proceedings of the 39th International Conference on Software Engineering*, pages 3–14. IEEE Press, 2017.
- [118] Xinli Yang, David Lo, Xin Xia, Lingfeng Bao, and Jianling Sun. Combining word embedding with information retrieval to recommend similar bug reports. In *Software Reliability Engineering (ISSRE), 2016 IEEE 27th International Symposium on*, pages 127–137. IEEE, 2016.
- [119] Bowen Xu, Zhenchang Xing, Xin Xia, and David Lo. Answerbot: automated generation of answer summary to developers’ technical questions. In *Proceedings of the 32nd IEEE/ACM International Conference on Automated Software Engineering*, pages 706–716. IEEE Press, 2017.
- [120] Zhuobing Han, Xiaohong Li, Zhenchang Xing, Hongtao Liu, and Zhiyong Feng. Learning to predict severity of software vulnerability using only vulnerability description. In *Software Maintenance and Evolution (ICSME), 2017 IEEE International Conference on*, pages 125–136. IEEE, 2017.
- [121] Zhuobing Han, Xiaohong Li, Hongtao Liu, Zhenchang Xing, and Zhiyong Feng. Deepweak: Reasoning common software weaknesses via knowledge graph embedding. In *2018 IEEE 25th International Conference on Software Analysis, Evolution and Reengineering (SANER)*, pages 456–466. IEEE, 2018.
- [122] Zhenlong Yuan, Yongqiang Lu, Zhaoguo Wang, and Yibo Xue. Droid-sec: deep learning in android malware detection. In *ACM SIGCOMM Computer Communication Review*, volume 44, pages 371–372. ACM, 2014.

- [123] Bojan Kolosnjaji, Apostolis Zarras, George Webster, and Claudia Eckert. Deep learning for classification of malware system call sequences. In *Australasian Joint Conference on Artificial Intelligence*, pages 137–149. Springer, 2016.
- [124] Guo Li, Haiyi Zhu, Tun Lu, Xianghua Ding, and Ning Gu. Is it good to be like wikipedia?: Exploring the trade-offs of introducing collaborative editing model to q&a sites. In *Proceedings of the 18th ACM Conference on Computer Supported Cooperative Work & Social Computing*, pages 1080–1091. ACM, 2015.
- [125] Lena Mamykina, Bella Manoim, Manas Mittal, George Hripcsak, and Björn Hartmann. Design lessons from the fastest q&a site in the west. In *SIGCHI*, pages 2857–2866. ACM, 2011.
- [126] Daniel Ortiz-Martinez, Ismael García-Varea, and Francisco Casacuberta. Thot: a toolkit to train phrase-based statistical translation models. *Tenth Machine Translation Summit. AAMT, Phuket, Thailand, September, 2005*.
- [127] Fernanda B Viegas, Martin Wattenberg, and Jonathan Feinberg. Participatory visualization with wordle. *Visualization and Computer Graphics, IEEE Transactions on*, 15(6):1137–1144, 2009.
- [128] Lasse Bergroth, Harri Hakonen, and Timo Raita. A survey of longest common subsequence algorithms. In *String Processing and Information Retrieval, 2000. SPIRE 2000. Proceedings. Seventh International Symposium on*, pages 39–48. IEEE, 2000.
- [129] Alex Graves, Abdel-rahman Mohamed, and Geoffrey Hinton. Speech recognition with deep recurrent neural networks. In *Acoustics, speech and signal processing (icassp), 2013 ieee international conference on*, pages 6645–6649. IEEE, 2013.
- [130] 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.
- [131] Paul J Werbos. Backpropagation through time: what it does and how to do it. *Proceedings of the IEEE*, 78(10):1550–1560, 1990.

- [132] Thang Luong, Ilya Sutskever, Quoc V. Le, Oriol Vinyals, and Wojciech Zaremba. Addressing the rare word problem in neural machine translation. In *Proceedings of the 53rd Annual Meeting of the Association for Computational Linguistics and the 7th International Joint Conference on Natural Language Processing of the Asian Federation of Natural Language Processing, ACL 2015, July 26-31, 2015, Beijing, China, Volume 1: Long Papers*, pages 11–19, 2015.
- [133] Guntis Barzdins, Steve Renals, and Didzis Gosko. Character-level neural translation for multilingual media monitoring in the summa project. *arXiv preprint arXiv:1604.01221*, 2016.
- [134] Martín Abadi, Ashish Agarwal, Paul Barham, Eugene Brevdo, Zhifeng Chen, Craig Citro, Greg S Corrado, Andy Davis, Jeffrey Dean, Matthieu Devin, et al. Tensorflow: Large-scale machine learning on heterogeneous distributed systems. *arXiv preprint arXiv:1603.04467*, 2016.
- [135] Robert Dale, Ilya Anisimoff, and George Narroway. Hoo 2012: A report on the preposition and determiner error correction shared task. In *Proceedings of the Seventh Workshop on Building Educational Applications Using NLP*, pages 54–62. Association for Computational Linguistics, 2012.
- [136] Keisuke Sakaguchi, Courtney Napoles, Matt Post, and Joel Tetreault. Reassessing the goals of grammatical error correction: Fluency instead of grammaticality. *Transactions of the Association for Computational Linguistics*, 4:169–182, 2016.
- [137] Courtney Napoles, Keisuke Sakaguchi, Matt Post, and Joel Tetreault. Ground truth for grammatical error correction metrics. In *Proceedings of the 53rd Annual Meeting of the Association for Computational Linguistics and the 7th International Joint Conference on Natural Language Processing*, volume 2, pages 588–593, 2015.
- [138] Courtney Napoles, Keisuke Sakaguchi, Matt Post, and Joel Tetreault. Gleu without tuning. *arXiv preprint arXiv:1605.02592*, 2016.

- [139] Kishore Papineni, Salim Roukos, Todd Ward, and Wei-Jing Zhu. Bleu: a method for automatic evaluation of machine translation. In *Proceedings of the 40th annual meeting on association for computational linguistics*, pages 311–318. Association for Computational Linguistics, 2002.
- [140] Courtney Napoles, Keisuke Sakaguchi, and Joel Tetreault. There’s no comparison: Reference-less evaluation metrics in grammatical error correction. *arXiv preprint arXiv:1610.02124*, 2016.
- [141] Ying Zhang, Stephan Vogel, and Alex Waibel. Interpreting bleu/nist scores: How much improvement do we need to have a better system? In *LREC*, 2004.
- [142] Lingfeng Bao, Jing Li, Zhenchang Xing, Xinyu Wang, Xin Xia, and Bo Zhou. Extracting and analyzing time-series hci data from screen-captured task videos. *Empirical Software Engineering*, pages 1–41, 2016.
- [143] Rakesh Agrawal, Ramakrishnan Srikant, et al. Fast algorithms for mining association rules. In *VLDB*, volume 1215, pages 487–499, 1994.
- [144] 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.
- [145] Mathieu Bastian, Sebastien Heymann, Mathieu Jacomy, et al. Gephi: an open source software for exploring and manipulating networks. *ICWSM*, 8:361–362, 2009.
- [146] Ulrik Brandes, Daniel Delling, Marco Gaertler, Robert Görke, Martin Hoefer, Zoran Nikoloski, and Dorothea Wagner. Maximizing modularity is hard. *arXiv preprint physics/0608255*, 2006.
- [147] Mathieu Jacomy, Sebastien Heymann, Tommaso Venturini, and Mathieu Bastian. Forceatlas2, a continuous graph layout algorithm for handy network visualization. *Medialab center of research*, 560, 2011.
- [148] Santo Fortunato. Community detection in graphs. *Physics Reports*, 486(3):75–174, 2010.

- [149] 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.
- [150] Chunyang Chen and Zhenchang Xing. Towards correlating search on google and asking on stack overflow. In *Computer Software and Applications Conference (COMPSAC), 2016 IEEE 40th Annual*, volume 1, pages 83–92. IEEE, 2016.
- [151] Google. Traffic from search engine robots. <https://support.google.com/analytics/answer/1315708?hl=en>, .
- [152] Google. Google trends. <https://www.google.com.sg/trends/>, .
- [153] John Brooke. Sus-a quick and dirty usability scale. *Usability evaluation in industry*, 189(194):4–7, 1996.
- [154] Wenliang Chen, Yue Zhang, and Min Zhang. Feature embedding for dependency parsing. In *Proceedings of the International Conference on Computational Linguistics*, 2014.
- [155] Tomas Mikolov, Wen-tau Yih, and Geoffrey Zweig. Linguistic regularities in continuous space word representations. In *HLT-NAACL*, pages 746–751, 2013.
- [156] 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.
- [157] Steven Bird. Nltk: the natural language toolkit. In *Proceedings of the COLING/ACL on Interactive presentation sessions*, pages 69–72. Association for Computational Linguistics, 2006.
- [158] 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*, pages 1378–1385. ACM, 2016.

- [159] Chunyang Chen and Zhenchang Xin. Towards correlating search on google and asking on stack overflow. In *40th Annual Computer Software and Applications Conference (COMPSAC)*, pages 83–92. IEEE, 2016.
- [160] Rudi L Cilibrasi and Paul MB Vitanyi. The google similarity distance. *TKDE*, 19(3):370–383, 2007.
- [161] Risto Gligorov, Warner ten Kate, Zharko Aleksovski, and Frank Van Harmelen. Using google distance to weight approximate ontology matches. In *WWW*, pages 767–776. ACM, 2007.
- [162] Christopher D Manning, Prabhakar Raghavan, Hinrich Schütze, et al. *Introduction to information retrieval*, volume 1. Cambridge university press Cambridge, 2008.
- [163] 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.
- [164] Google. Google analytics policy. <https://support.google.com/analytics/answer/1315708?hl=en>, 2016.
- [165] Mathias Creutz and Krista Lagus. Unsupervised morpheme segmentation and morphology induction from text corpora using morfessor 1.1. *Publications Comput. Inf. Sci., Helsinki Univ. Technol., Tech. Rep. A*, 81:2005, 2005.
- [166] Jan A Botha and Phil Blunsom. Compositional morphology for word representations and language modelling. In *ICML*, pages 1899–1907, 2014.
- [167] Thang Luong, Richard Socher, and Christopher D Manning. Better word representations with recursive neural networks for morphology. In *CoNLL*, pages 104–113, 2013.
- [168] Radu Soricut and Franz Och. Unsupervised morphology induction using word embeddings. In *Proc. NAACL*, 2015.

- [169] Martin F Porter. An algorithm for suffix stripping. *Program*, 14(3):130–137, 1980.
- [170] Wordnet lemmatization. http://www.nltk.org/_modules/nltk/stem/wordnet.html.
- [171] Youngja Park, Siddharth Patwardhan, Karthik Visweswariah, and Stephen C Gates. An empirical analysis of word error rate and keyword error rate. In *INTERSPEECH*, pages 2070–2073, 2008.
- [172] Stack overflow data dump. <https://archive.org/details/stackexchange>. Accessed: 2016-02-20.
- [173] Wikipedia data dump. <https://dumps.wikimedia.org/enwiki/latest/>, . Accessed: 2016-02-20.
- [174] Simone Paolo Ponzetto and Michael Strube. Knowledge derived from wikipedia for computing semantic relatedness. *J. Artif. Intell. Res.(JAIR)*, 30:181–212, 2007.
- [175] Simone Paolo Ponzetto and Michael Strube. Deriving a large scale taxonomy from wikipedia. In *AAAI*, volume 7, pages 1440–1445, 2007.
- [176] Tomas Mikolov, Kai Chen, Greg Corrado, and Jeffrey Dean. Efficient estimation of word representations in vector space. *arXiv preprint arXiv:1301.3781*, 2013.
- [177] Ian Jolliffe. *Principal component analysis*. Wiley Online Library, 2002.
- [178] Vladimir I Levenshtein. Binary codes capable of correcting deletions, insertions, and reversals. In *Soviet physics doklady*, volume 10, pages 707–710, 1966.
- [179] Fred J Damerau. A technique for computer detection and correction of spelling errors. *Communications of the ACM*, 7(3):171–176, 1964.
- [180] Abbreviations in wikipedia. https://en.wikipedia.org/wiki/List_of_computing_and_IT_abbreviations, . Accessed: 2016-06-20.

- [181] Tag synonyms in stack overflow. <http://stackoverflow.com/tags/synonyms>. Accessed: 2016-06-20.
- [182] Michael Gegick, Pete Rotella, and Tao Xie. Identifying security bug reports via text mining: An industrial case study. In *Mining software repositories (MSR), 2010 7th IEEE working conference on*, pages 11–20. IEEE, 2010.
- [183] Gayane Petrosyan, Martin P Robillard, and Renato De Mori. Discovering information explaining api types using text classification. In *Proceedings of the 37th International Conference on Software Engineering-Volume 1*, pages 869–879. IEEE Press, 2015.
- [184] Erik Linstead, Paul Rigor, Sushil Bajracharya, Cristina Lopes, and Pierre Baldi. Mining concepts from code with probabilistic topic models. In *Proceedings of the twenty-second IEEE/ACM international conference on Automated software engineering*, pages 461–464. ACM, 2007.
- [185] Lori Pollock, K Vijay-Shanker, Emily Hill, Giriprasad Sridhara, and David Shepherd. Natural language-based software analyses and tools for software maintenance. In *Software Engineering*, pages 94–125. Springer, 2013.
- [186] Anthony CW Finkelstein, Dov Gabbay, Anthony Hunter, Jeff Kramer, and Bashar Nuseibeh. Inconsistency handling in multiperspective specifications. *Software Engineering, IEEE Transactions on*, 20(8):569–578, 1994.
- [187] 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.
- [188] Raphael Hoffmann, James Fogarty, and Daniel S Weld. Assieme: finding and leveraging implicit references in a web search interface for programmers. In *Proceedings of the 20th annual ACM symposium on User interface software and technology*, pages 13–22. ACM, 2007.
- [189] Hao Zhong, Lu Zhang, Tao Xie, and Hong Mei. Inferring resource specifications from natural language api documentation. In *Proceedings of the 2009 IEEE/ACM*

- International Conference on Automated Software Engineering*, pages 307–318. IEEE Computer Society, 2009.
- [190] Rila Mandala, Tokunaga Takenobu, and Tanaka Hozumi. The use of wordnet in information retrieval. In *Use of WordNet in Natural Language Processing Systems: Proceedings of the Conference*, pages 31–37, 1998.
- [191] Angelos Hliaoutakis, Giannis Varelakis, Epimenidis Voutsakis, Euripides GM Petrakis, and Evangelos Milios. Information retrieval by semantic similarity. *International journal on semantic Web and information systems (IJSWIS)*, 2(3):55–73, 2006.
- [192] Wilbert O Galitz. *The essential guide to user interface design: an introduction to GUI design principles and techniques*. John Wiley & Sons, 2007.
- [193] Y Wang B Hong. Matters of design. In *Commun. ACM*. Citeseer, 2011.
- [194] Chunyang Chen, Ting Su, Guozhu Meng, Zhenchang Xing, and Yang Liu. From ui design image to gui skeleton: A neural machine translator to bootstrap mobile gui implementation. In *Proceedings of the 40th International Conference on Software Engineering*, pages 665–676. IEEE Press, 2018.

Appendices

All works presented in this thesis are either published or accepted in the following international conferences. As shown in Fig. 1.2, the work in Chapter 3 is from *CSCW 2018*. The work in Chapter 4 is mainly from the paper in *EMSE 2016* and *ICSME 2016*; The work in Chapter 5 is from the papers in *SANER 2016* and *ASE tool 2016*; The work in Chapter 6 is from the paper in *ICSE 2017*;

All publications of the Ph.D candidate are listed as follows, and the publications that are not included in the thesis are marked with an asterisk:

1. * **Chunyang Chen**, Ting Su, Guozhu Meng, Zhenchang Xing, Yang Liu, “*From UI Design Image to GUI Skeleton: A Neural Machine Translator to Bootstrap Mobile GUI Implementation*”, The 40th International Conference on Software Engineering (ICSE 2018), Gothenburg, Sweden.
2. **Chunyang Chen**, Zhenchang Xing, Yang Liu, “*By the Community & For the Community: A Deep Learning Approach to Assist Collaborative Editing in Q&A Sites*”, The 21st ACM Conference on Computer-Supported Cooperative Work and Social Computing (CSCW 2018), pp. 32:1-32:21, New York, USA
3. **Chunyang Chen**, Zhenchang Xing, Ximing Wang, “*Unsupervised Software-Specific Morphological Forms Inference from Informal Discussions*”, The 39th International Conference on Software Engineering (ICSE 2017), pp. 450-461, Buenos Aires, Argentina.
4. * Guibin Chen, **Chunyang Chen**, Zhenchang Xing, Bowen Xu, “*Learning a Dual-Language Vector Space for Domain-Specific Cross-Lingual Question Retrieval*”, The 31st IEEE/ACM International Conference on Automated Software Engineering (ASE 2016), pp. 744-755, Singapore.
5. **Chunyang Chen**, Zhenchang Xing, “*SimilarTech: Automatically Recommend Analogical Libraries across Different Programming Languages*”, The 31st IEEE/ACM International Conference on Automated Software Engineering (ASE 2016), pp. 834-839, Singapore.

-
6. **Chunyang Chen**, Zhenchang Xing, Lei Han, “*TechLand: Assisting Technology Landscape Inquiries with Insights from Stack Overflow*”, The 32nd IEEE International Conference on Software Maintenance and Evolution (ICSME 2016), pp. 356-366, Raleigh, North Carolina, USA.
 7. **Chunyang Chen**, Zhenchang Xing, “*Mining Technology Landscape from Stack Overflow*”, The 10th ACM/IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM 2016), pp. 14:1-14:10, Ciudad Real, Spain.
 8. **Chunyang Chen**, Sa Gao, Zhenchang Xing, “*Mining Analogical Libraries in Q&A Discussions — Incorporating Relational and Categorical Knowledge into Word Embedding*”, The 23rd IEEE International Conference on Software Analysis, Evolution, and Reengineering (SANER 2016), pp. 338-348, Osaka, Japan.
 9. * **Chunyang Chen**, Zhenchang Xing, “*Towards Correlating Search on Google and Asking on Stack Overflow*”, The 40th IEEE Computer Society International Conference on Computers, Software & Applications (COMPSAC 2016), pp. 83-92, Atlanta, Georgia, USA.