Supporting information needs of developers through web Q&A discussions

Li, Jing

2018

Li, J. (2018). Supporting information needs of developers through web Q&A discussions.
Doctoral thesis, Nanyang Technological University, Singapore.

<http://hdl.handle.net/10356/74202>

<https://doi.org/10.32657/10356/74202>

**NANYANG
TECHNOLOGICAL
UNIVERSITY**

SINGAPORE

**SUPPORTING INFORMATION NEEDS OF
DEVELOPERS THROUGH WEB Q&A
DISCUSSIONS**

LI JING

SCHOOL OF COMPUTER SCIENCES AND ENGINEERING

2018

Supporting Information Needs of Developers through Web Q&A Discussions

LI JING

School of Computer Sciences and Engineering

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

2018

Abstract

Programming is evolving because of the prevalence of the Web. Nowadays, it is a common activity that developers search the Web to find information in order to solve the problems they encounter while working on software development tasks. However, existing studies investigated the information needs of developers on the Web via qualitative analysis and questionnaire survey. Unfortunately, little is known about the developers' micro-level information behaviors and needs on the Web during software development. For example, how often did the developers refine existing queries and/or create new queries? and how many web pages were opened after a search?

To fill this gap, we conducted an empirical study to investigate the strategies that how developers seek and use web resources at the micro-level. The empirical study revealed *three key insights*: *First*, developers might have an incomplete or even incorrect understanding of their needs; *Second*, there is a gap between the producers and consumers of software documentation; *Third*, many important pieces of information that developers need are explicitly undocumented in software documentation. These insights motivated further studies of supporting developers' information needs. More specifically, the contributions of this thesis are:

- (1) Understanding information needs of developers: We developed a video scraping tool to automatically extract developers' behavioral data from the task videos. We conducted a micro-level quantitative analysis of the developers' information, including patterns of keyword sources, keyword refinement, web pages visited, context switching, and information flow. The outcomes of this micro-level quantitative analysis provided three important insights for supporting information needs of developers.

- (2) Discovering learning resources: To bridge the information gap in the first insight, we developed our LinkLive technique to recommend more correlated learning resources when developers know less. LinkLive uses multiple features, including hyperlink co-occurrences in web Q&A discussions, locations (*e.g.*, question, answer, or comment) in which hyperlinks are referenced, and votes for posts/comments in which hyperlinks are referenced. A large-scale evaluation shows that our technique recommends correlated web resources with satisfactory precision and recall in an open setting.
- (3) Answering programming questions: To bridge the information gap in the second insight, we proposed a novel *deep-learning-to-answer* framework, named QDLinker, for answering programming questions with software documentation. QDLinker leverages the large volume of discussions in Community-based Question Answering (CQA) to bridge the semantic gap between programmers' questions and software documentation. Through extensive experiments, we show that QDLinker significantly outperforms the baselines based on traditional retrieval models and Web search services dedicated for software documentation.
- (4) Distilling crowdsourced API negative caveats: To bridge the information gap in the third insight, we proposed **DISCA**, a novel approach to automatically distilling desirable Application Program Interface (API) negative caveats from unstructured web Q&A discussions. The quantitative and qualitative evaluations show that **DISCA** can greatly augment the official API documentation.

Acknowledgments

First and foremost, I would like give my sincere thanks to my advisors Professor Sun Aixin and Professor Xing Zhenchang for the continuous support of my Ph.D study. Before Professor Xing moved to the Australian National University, I got the chance from him to pursuit my Ph.D in Nanyang Technological University. I appreciate all his contributions of ideas, time and funding to make my Ph.D experience stimulating. I am grateful to Professor Sun for his continuous support of the second half of my Ph.D study. I am really thankful for his weekly meeting and carefully revising my submissions. His patient guidance helped me in all the time of my research and writing of this thesis. It is a great honor for me to pursuit my Ph.D under supervision of the two advisors.

I would like to thank my thesis committee: Professor Lo David and Professor Cong Gao, for their insightful comments and encouragement, but also for the critical questions which prompted me to widen my research from various perspectives.

I am fortunate to have had great colleagues and friends throughout the four years. I would like to thank my group members at the Data Management and Analytics Lab: Dr. Han Jialong, Dr. Ma Zongyang, Miss Zheng Xin, Mr. Han Jianglei, Mr. Phan Cong Minh, Miss Grace Lee, Mr. Lin Ting, Mr. Huang Keke, Mr. Chen Zhe, Mr. Wang Yequan and Mr. Han Peng for making the group a vibrant and friendly one. I would like to express appreciation for my friends and colleagues at the Cyber Security Lab for their support: Dr. Liu Yi, Dr. Bao Lingfeng, Dr. Ye Deheng, Miss Zhao Xuejiao, Mr. Chen Chunyang, and Mr. Gao Sa. At NTU, there are numerous other friends that I am afraid I cannot list exhaustively here – I owe all of them a big thanks!

Ph.D thesis acknowledgment usually does not include pre-college teachers, which I think is not a good tradition. My sincere thanks go to all the teachers I met in my pre-college life. I would list three ones: Mr. He Lihua, who was my first teacher in primary school, encouraged me to step out the small village in my hometown; Mr.

Lu Jun, who was the head teacher of my middle school, encouraged me to step out the town; Mr. Tian Wei, who taught me high-school math, encouraged me to go to top university. Without them, I would never get the chance to embark on a research journey.

Last but not the least, I would like to thank my family. I would like to express my deepest respect and gratitude to my parents, for their unconditional love, trust, and support! I am also indebted to my sister, who supported my life in general and took care of our parents when I was studying abroad. Their care and company are precious treasures in my life. This thesis is dedicated to them.

LI Jing
December 2017
Singapore

Contents

Abstract	i
Acknowledgments	iii
Contents	viii
List of Figures	xi
List of Tables	xiii
List of Abbreviations	xiv
List of Notations	xv
1 Introduction	1
1.1 Motivation and Objectives	1
1.2 Challenges and Contributions	5
1.3 Thesis Outline	7
2 Literature Review	9
2.1 Information Needs of Developers	9
2.2 Role of The Web in Software Development	12
2.3 Recommendation Systems in Software Engineering	14
2.4 Summary	16
3 Empirical Study: Understanding Information Needs of Developers Through Web Usage	18
3.1 Background and Motivation	18
3.2 Data Collection	20
3.2.1 Experimental Design	20
3.2.2 Video Scraping	21
3.3 Micro-level Study of Developers' Information Needs	22
3.3.1 Analysis of Search Queries	23

3.3.2	Analysis of Web Pages Visited	26
3.3.3	Patterns of Seeking-using Web Information	28
3.4	Key Insights	37
3.4.1	Need to Find More When You Know Less: Developers Might Have An Incomplete or Even Incorrect Understanding of Their Needs	37
3.4.2	There Is A Gap Between The Producers And Consumers of Soft- ware Documentation	38
3.4.3	Many Important Pieces of Information That Developers Need Are Explicitly Undocumented in Software Documentation	39
3.5	Summary	40
4	Discovering Learning Resources from Web Q&A Discussions	41
4.1	Background and Motivation	41
4.2	An Exploratory Study of Hyperlinks in Stack Overflow	45
4.2.1	Dataset	46
4.2.2	What and Where About “Referenced”	46
4.2.3	Hyperlink Reference and Co-occurrence Frequency	46
4.2.4	Hyperlink-Vote Correlation	48
4.3	The LinkLive Approach	49
4.3.1	Hyperlink Associative Network	49
4.3.2	Associative-Edge Weight Computation	52
4.3.3	ROI Recommendation	54
4.4	The LinkLive Tool	54
4.5	Evaluation of the LinkLive Recommendation	54
4.5.1	Experiment Setup	55
4.5.2	The Accuracy of LinkLive Recommendation	56
4.5.3	The Reliability of LinkLive Recommendation	60
4.6	User Study	61
4.6.1	Study Design	61
4.6.2	Perceived Helpfulness	62

4.6.3	Perceived Diversity	64
4.6.4	Threats to Validity	65
4.7	Discussion	66
4.8	Summary	67
5	Learning to Answer Programming Questions with Software Documentation via Web Q&A Context Embedding	68
5.1	Background and Motivation	69
5.2	Related Work	72
5.3	Deep Learning to Answer	74
5.3.1	Social Context Embedding	76
5.3.2	Candidate Document Generation	78
5.3.3	Four-layer Deep Neural Network	79
5.3.4	Learning a Ranker	83
5.3.5	Summary and Complexity Analysis	84
5.4	Empirical Evaluation	86
5.4.1	Experimental Setting	86
5.4.2	Performance Comparison	88
5.4.3	Impacts of Factors on Performance	90
5.5	User Study	92
5.5.1	Evaluation Setup	92
5.5.2	Evaluation Results	94
5.6	Summary	95
6	Distilling Crowdsourced API Negative Caveats from Web Q&A Discussions to Augment Software Documentation	97
6.1	Background and Motivation	97
6.2	Problem Definition	101
6.3	The Disca Approach	103
6.3.1	Input Data	103
6.3.2	Selecting Candidate Sentences	104
6.3.3	Filtering Out Context-Dependent Sentences	105

6.3.4	Identifying Prominent Domain-specific Terms	106
6.3.5	Discovering Semantically Diverse Aspects	106
6.3.6	Selecting API Negative Caveats	107
6.4	Evaluation	110
6.4.1	Experimental Settings	111
6.4.2	Quantitative Evaluation	115
6.4.3	Qualitative Analysis	118
6.5	Tool Implementation	121
6.6	Threats to Validity	123
6.7	Related Work	124
6.7.1	Crowdsourced knowledge in software development	124
6.7.2	App review opinion mining	125
6.7.3	Automatic text summarization	126
6.8	Summary	127
7	Conclusion and Future Work	129
7.1	Conclusion	129
7.2	Future Work	130
A	Appendix	132
A.1	Evaluation results of scvRipper	132
A.2	Predefined patterns for filtering out context-dependent sentences	134
	Publications	136
	References	139

List of Figures

1.1	The thesis overview	4
1.2	The thesis outline	7
3.1	Extracting time-series HCI data from screen-captured videos by <i>scoRipper</i>	23
3.2	The statistics of new queries and query refinements	26
3.3	The statistics of opening a specific number of Web pages	28
3.4	Statistics of unique URLs and URL switchings	29
3.5	Statistics of context switchings in every 10 minutes	30
3.6	An illustrative example of search sessions	33
3.7	Heat map of the feature values of the 4 search-session clusters	34
3.8	Timeline plot of search-session types	36
4.1	An example of web linked resources in Stack Overflow. Hyperlinks are underlined.	43
4.2	The hyperlink and domain citation distribution	47
4.3	Distribution of number of hyperlinks per discussion thread	48
4.4	Hyperlink co-occurrence graph	50
4.5	The LinkLive tool	55
4.6	Overall performance of the LinkLive recommendation	57
4.7	Performance on the three testing subsets	60
4.8	Average recommendation helpfulness score at different citation levels . .	63
4.9	Average recommendation helpfulness score for different participants . .	64
4.10	Distribution of categories of recommended web resources	64
5.1	Distribution of links to Java documentation per best answer, among 45, 288 best answers from Stack Overflow	70

5.2	Overview of QDLinker. It directly links programmer’s question to formal documentation through embedding the semantic context in CQA. Before QDLinker, the semantic links between questions and software documentation are established through the CQA community.	71
5.3	The architecture of our proposed approach	75
5.4	Training word embedding example with the skip-gram model	76
5.5	A 2D projection of embedding natural language terms and API documentation using PCA (API documentation in bold font and natural language terms in non-bold font)	78
5.6	An example of extracting query-documentation pair from discussion thread in Stack Overflow	83
5.7	The flow diagram of the proposed QDLinker. The middle part is to learn a model for extracting abstract representation for question-documentation pair. The left part aims to learn a ranker. The right part is to answer a new question with the learned models.	84
5.8	Impact of dimensionality of word embedding	90
5.9	Performance in MAP with different layer sizes	91
6.1	Examples of three official API documents	99
6.2	Overview of the DISCA framework	103
6.3	Three examples of dependency parsing for sentences: “ <i>HashMap doesn’t define the order of iteration over the elements</i> ”, “ <i>Don’t use a HashMap with multiple threads</i> ” and “ <i>JSONObject does not have too much additional overhead on top of a HashMap</i> ”. Note that the two examples on the left hand side are selected as candidate sentences in this study.	105
6.4	Term Communities shown in different colors identified from the term co-occurrence graph of java.util.HashMap	108
6.5	Selected API negative caveats for the term community “ <i>multiple thread synchronized</i> ”	111
6.6	nDCG values of the five methods at different ranking positions {1, 2, 3, 4, 5, all}	116

6.7	DISCA screenshots (best viewed in color). The main GUI consists of five components: ① The query input panel. ② Interactive modes: Foam-Tree, Circles, and Fisheye. ③ Term community, representing a semantic aspect of an API. ④ Link to the original post on Stack Overflow. ⑤ The sentence detailing the API caveat.	122
A.1	The screenshot of scvRipper	132

List of Tables

3.1	Statistics of distinct keywords and keyword sources	24
3.2	Most-used keywords in the two tasks	25
3.3	The top three most-visited Web sites of 7 Web categories	27
3.4	Markov transition matrices in task 1	31
3.5	Markov transition matrices in task 2	31
4.1	Top-20 most referenced domains	45
4.2	Statistics of questions/answers/comments having hyperlinks	47
4.3	Statistics of the sources of distinct hyperlinks	47
4.4	The notations of this chapter	49
4.5	Number of seed ROIs and citations at different citation levels	58
4.6	Precision at different citation levels	59
4.7	Recall at different citation levels	59
5.1	Summary of the number of threads used in each task	87
5.2	Performance ($P@k$, $R@k$, MAP and MRR) for different methods. The best performance is highlighted in bold face. † indicates that the differences between the result of QDLinker and other models are statistically significant with $p < 0.05$ under t -test.	89
5.3	Performance($HR@k$) for different methods	89
5.4	Results on additional content features	90

5.5	Human evaluation for Google search and QDLinker (<i>BA</i> : the number of ground truth links in best answer. <i>FR</i> : the rank of the first relevant documentation. <i>P@5</i> and <i>P@10</i> : precision of the first 5 and 10 results. “ A ” indicates Java API documentatins. “ S ” indicates Java language specification. “ T ” indicates Java tutorial. The boldface documentation indicate the ground truth documentation in best answers. † indicates the differences between QDLinker and Google are significant with $p < 0.05$ under <i>t</i> -test).	93
6.1	Example API negative caveats extracted from Stack Overflow	98
6.2	The statistics of Java API types used in evaluations. “ <i>Cand_x</i> ” and “ DISCA ” denote number of candidate API negative caveats and number of negative caveats mined by DISCA , respectively	112
6.3	Agreement between annotators, by evaluating the summary of one annotator against the summaries of the other two annotators on three ROUGE measures	114
6.4	Performance of the five methods on ROUGE measures. The best performance is highlighted in bold face. † indicates that the improvements made by DISCA over the best baseline is statistically significant under paired <i>t</i> -test with $p \leq 0.05$.	115
6.5	ROUGE scores obtained by DISCA on different Java API types	117
6.6	Results of inter-method comparison, based on ROUGE measures	117
6.7	The three columns show the numbers of negative caveats that are mentioned in official documentation, distilled by DISCA , and the matched between the two	119
6.8	Examples of API negative caveats mined by DISCA	120

List of Abbreviations

Q&A	Question and Answering
CQA	Community-based Question Answering
API	Application Program Interface
URL	Uniform Resource Locator
OCR	Optical Character Recognition
HCI	Human Computer Interaction
IDE	Integrated Development Environment
HAN	Hyperlink Associative Network
KS-test	KolmogorovSmirnov test
ROI	Resource-of-Interest
DNN	Deep Neural Network
PCA	Principal Component Analysis
CNN	Convolutional Neural Network
ROUGE	Recall-Oriented Understudy for Gisting Evaluation
nDCG	Normalized Discounted Cumulative Gain
LDA	Latent Dirichlet Allocation
MMR	Maximal Marginal Relevance

List of Notations

h_s	A seed ROI
h_t	A candidate ROI
$\mathcal{G}_{h_s} = (V, E)$	Hyperlink co-occurrence graph for h_s , Vertex V , Edge E
$HAN = (H, X)$	Hyperlink Associative Network. $HAN = \cup \mathcal{G}_{h_s}$
$e_{s,t}$	The edge from node h_s to node h_t
$\omega_{s,t}$	The weight of edge $e_{s,t}$ in HAN
$S_{e_{s,t}}$	The score of edge $e_{s,t}$ in E
$S'_{e_{s,t}}$	The normalized score of edge $e_{s,t}$ in E
$\mathbf{X}_d$	Software documentation vector
$\mathbf{X}_q$	Query vector
$\mathbf{C}_q \in \mathbb{R}^{n \times (s-m+1)}$	Feature maps for queries
$\mathbf{C}_d \in \mathbb{R}^{n \times (s-m+1)}$	Feature maps for documents
$\mathbf{f}_{cn}$	Content features
$Cand_x$	Candidate API negative caveats for API type x
$\mathcal{A}_x$	The semantically diverse and non-redundant sentences that cover the most prominent domain-specific terms related to the usage issues of API type x

1 Introduction

You never fail until you stop trying.

— Albert Einstein

The way software developers develop software has been changed because of the large amount of information available on the *Web*. Looking for knowledge from the Web to adapt to their *information needs*, is a key factor in improving developers' effectiveness on software development and maintenance tasks. Understanding *information needs* of developers can motivate the design of current software development and maintenance tools. The whole line of research contained in this thesis was inspired by an empirical study, which concerns how developers seek and use Web resources in two software development tasks. In this Chapter, we first introduce research motivation and goals, then detail the challenges and thesis contributions.

1.1 Motivation and Objectives

Programming is evolving because of the prevalence of the Web [Brandt et al., 2010, Zalgalsky et al., 2012, Nasehi et al., 2012]. Nowadays, it is a common activity that developers search the Web to find information in order to solve the problems they encounter while working on software development tasks. Some studies have investigated how developers seek, relate and collect relevant information during software maintenance tasks [Ko et al., 2006, Ko et al., 2007]. Brandt *et al.* [Brandt et al., 2010] and Li *et al.* [Li et al., 2013] have investigated what online web resources developers seek, and when and how they seek and use web resources. These studies have confirmed that effectively improving developers' effectiveness hinges on the ability to quickly and accurately find desired knowledge.

Searching on the Web is the most common activity for software engineers [Buse and Zimmermann, 2012, Begel and Zimmermann, 2013, Treude et al., 2011]. Thus, the web usage would be helpful to understand information needs and behaviors of developers. Existing studies investigated information needs of developers at a high-level process-oriented analysis [Li et al., 2013]. Meanwhile, most of them were based on qualitative analysis and questionnaire survey [Begel and Zimmermann, 2013, Buse and Zimmermann, 2012]. However, little is known about the developers' micro-level information behaviors and needs during software development. For example, how often did the developers refine existing queries and/or create new queries? and how many web pages were opened after a search?

On the other hand, conducting micro-level analysis of information needs of developers is no simple task. Studies on developers' information needs often analyze the task videos recorded using screen-capture software during the task. Such task videos provide rich information about the progression of the task, the application usage, and the data accessed and used by the developers. Performing a micro-level quantitative analysis of such task videos is prohibitively expensive, as this requires transcribing and coding unstructured video data. Studies on human behavior research report that the ratio of recoding time and analysis time can be 1:15-100 [Sanderson and Fisher, 1994], depending on the details and granularity of the information to be collected.

To fill this gap, we conducted an empirical study to investigate the strategies that how developers seek and use web resources.

Research Objective 1: Understanding information needs of developers.

Develop a video scraping tool to automatically extract data and analyze developers' information needs through web usage at the micro-level.

To achieve this objective, we first will develop a video scraping tool for automatically transcribing and coding screen-captured task videos. We will investigate a micro-level quantitative analysis of 29-hours task videos collected in our previous study [Li et al., 2013]. We will analyze search queries, web pages visited and the patterns of seeking-using web information during software development.

The achievements of research objective 1 provide following **three insights**:

Insight 1: Need to find more when you know less: developers might have an incomplete or even incorrect understanding of their needs;

Insight 2: There is a gap between the producers and consumers of software documentation;

Insight 3: Many important pieces of information that developers need are explicitly undocumented in software documentation.

The three insights motivated further studies of supporting developers' information needs.

Research Objective 2: Discovering learning resources.

Design model to recommend learning resources from web Q&A discussions.

Insight 1 suggests that developers might have an incomplete or even incorrect understanding of their needs. There are many situations in which developers only know partial topics they are looking for, but at the same time is willing to explore correlated web resources to extend their knowledge or to satisfy their curiosity. Stack Overflow¹ has been providing question and answering service for 9 years. It has become a tremendous knowledge repository for developers' thoughts and practices. The ultimate objective is to recommend the top- k most correlated learning resources when developers know less using Stack Overflow dataset.

Research Objective 3: Answering programming questions.

Design model to bridge the gap between the content of software documentation and intent of developers.

Insight 2 reveals that this is a gap between the producers and consumers of software documentation. These official documents provide comprehensive coverage but not aim for specific programming tasks or use cases. Programmers often face very specific issues which are not explicitly stated in software documentation. We observe from Stack Overflow that the best answers to programmers' questions often contain links to formal documentation. Thus, we will design a model to answer programming questions with software documentation. The idea is to leverage the large volume of discussions in Community-based Question Answering (CQA) to bridge the semantic gap between programmers' questions and software documentation.

¹<https://stackoverflow.com/>

Figure 1.1: The thesis overview

Research Objective 4: Distilling crowdsourced API negative caveats.

Design model to distill overlooked but important caveats to augment software documentation.

Insight 3 reveals that many important pieces of information that developers need are explicitly undocumented in software documentation. Especially, negative caveats of API are about “how not to use an API”, which are often absent from the official API documentation. When these caveats are overlooked, programming errors may emerge from misusing APIs, leading to heavy discussions on Q&A websites like Stack Overflow. If the overlooked caveats could be mined from these discussions, they would be beneficial for programmers to avoid misuse of APIs. Thus, we will investigate the approach that automatically distills desirable API negative caveats from unstructured Q&A discussions.

In summary, Figure 1.1 shows the overview of thesis. The research objectives are divided into two parts: empirical study that aims to understand how developers seek and use web information, and three objectives that aim to support developers’ information needs.

1.2 Challenges and Contributions

These research objectives in Section 1.1 motivate the main work in this thesis. It is not-trivial to achieve the aforementioned objectives since these exist many technical challenges. We list the following challenges and contributions for each research objective, respectively.

(1) Challenges and contributions for research objective 1 (Understanding information needs of developers)

- **challenge 1.1: How to transcribe video data.** Performing a micro-level quantitative analysis of information needs of developers is prohibitively expensive, as this requires transcribing and coding screen-captured task videos.
- **contribution 1.1: Video scraping tool.** We developed a video scraping tool to automatically extract developers' behavioral data from the task videos, including application usage (*e.g.*, Eclipse, web browser) and application data (*e.g.*, search query, URL, code, exception message).
- **contribution 1.2: Micro-level studies of understanding information needs of developers.** We conducted a micro-level quantitative analysis of the developers' information, including patterns of keyword sources, keyword refinement, web pages visited, context switching, and information flow.

(2) Challenges and contributions for research objective 2 (Discovering learning resources):

- **challenge 2.1: Correlated learning resources may not have similar content.** Search-based methods often cannot help developers discover *correlated and new* web resources [Cooley et al., 1997], because search engines generally employ keyword matching or rely on certain content similarity of web resources. Correlated web resources, however, may not have similar content, for example, Singleton pattern², Abstract Factory pattern³, and Double-checked locking⁴. Further-

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

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

⁴https://en.wikipedia.org/wiki/Double-checked_locking

more, when developers have no or little knowledge about the correlated information they may be interested in, it is difficult for them to formulate an effective search query.

- **challenge 2.2: How to turn data into wisdom.** Wisdom on what web resources are highly-recognized by the community and are pertinent to the hyperlinks at hand is buried in the sea of discussion threads.
- **contribution 2.1: Web resource dissemination patterns.** We conducted a large-scale exploratory study of 5.5 millions hyperlinks referenced in Stack Overflow, to investigate its hyperlink dissemination patterns.
- **contribution 2.2: LinkLive technique.** We developed our LinkLive technique that uses multiple features, including hyperlink co-occurrences in Q&A discussions, locations (*e.g.*, question, answer, or comment) in which hyperlinks are referenced, and votes for posts/comments in which hyperlinks are referenced. We implemented a proof-of-concept tool LinkLive, which is implemented as a web browser add-on, based on GreaseMonkey/TamperMonkey technique.

(3) Challenges and contributions for research objective 3 (Answering programming questions):

- **challenge 3.1: Lexical gap between software documentation producers and consumers.** These official documents provide comprehensive coverage but not aim for specific programming tasks or use cases. Often there is a mismatch between the documentation and questions encountered in specific programming tasks because of different wordings.
- **contribution 3.1: An effective answering system.** we proposed a novel *deep learning to answer* framework, named QDLinker, for answering programming questions with software documentation. QDLinker learns from the large volume of discussions in Community-based Question Answering (CQA) to bridge the semantic gap between programmers' questions and software documentation.

Figure 1.2: The thesis outline

(4) Challenges and contributions for research objective 4 (Distilling crowdsourced API negative caveats):

- **challenge 4.1: Q&A discussions are context-dependent and redundant.** Sentences in a discussion often reference to other part(s) of the discussion. The distilled API negative caveats should be context-independent, so that programmers know a negative caveat without having to refer to the original discussion. In a large volume of informal discussion, the same API negative caveat may be mentioned many times but in different wording. A good approach should avoid selecting other sentences which are semantically redundant.
- **contribution 4.1: DISCA model.** We proposed **DISCA**, a novel approach to automatically **DISTILLING** desirable API negative **CAVEATS** from unstructured Q&A discussions. Through sentence selection and prominent term clustering, **DISCA** ensures the distilled caveats are *context-independent, prominent, semantically diverse and non-redundant*.

1.3 Thesis Outline

Figure 1.2 shows the roadmap for this dissertation, which is organized as follows:

Chapter 1 has introduced the motivation and research objectives.

Chapter 2 surveys related work.

Chapter 3 conducts an empirical study towards understanding information needs of developers through web usage.

Chapter 4 proposes LinkLive technique to discover learning resources from web Q&A discussions.

Chapter 5 proposes QDLinker to answer programming questions with software documentation via web Q&A context embedding.

Chapter 6 proposes DISCA to distill crowdsourced API negative caveats from web Q&A discussions to augment software documentation.

Chapter 7 summarizes our work, points out its limitations, and draws a research agenda.

Literature Review

2

All truths are easy to understand once they are discovered; the point is to discover them.

— Galileo Galilei

Related work for this thesis falls into three broad areas. First, researchers have long studied developers' behaviors in software development. Program comprehension is a key developer activity during software development, evolution and maintenance. Developers always search and use the Web resources during software development. Understanding developers' information needs is crucial for designing new tools to monitor, diagnose, and react to developers' behavior in software development. Second, Web resources have become key information resources for software development. The Web is fundamentally changing programming. We then summarize the studies about role of the Web in software development. Last, despite steady advancement in the state of the art, software development remains a challenging and knowledge-intensive activity. Recommendation systems are now emerging to assist software developers in various activities. We summarized the studies on helping software developers with recommendation systems.

2.1 Information Needs of Developers

Today, the software systems are more and more complex. This brings the challenges that developers need to read and understand amount of source code. For years, researchers have been attempting to reveal that how developers comprehend programs during software development, maintenance and evolution.

Von *et al.* [[Von Mayrhauser and Vans, 1995](#)] surveyed the knowledge in this area by comparing six program comprehension models: the Letovsky (1986) model; the

Shneiderman and Mayer (1979) model; the Brooks (1983) model; Soloway, Adelson and Ehrlich's (1988) top-down model; Pennington's (1987) bottom-up model; and the integrated metamodel of von Mayrhauser and Vans (1994). They found that these general models can foster a complete understanding of a piece of code, they may not always apply to specialized tasks that more efficiently employ strategies geared toward partial understanding.

Developers can more easily comprehend the source code entities if there are some short textual descriptions. Based on automated text summarization technology, Haiduc *et al.* [Haiduc *et al.*, 2010] proposed a framework to automatically annotate such textual descriptions. In a large software development project, developers always need to make some changes or modifications of source code that they are unfamiliar with. DeLine *et al.* [DeLine *et al.*, 2005] presented Team Tracks to assist developers in program comprehension. This system shows the source code navigation patterns of fellow development team members.

Wettel *et al.* [Wettel and Lanza, 2007] proposed a 3D visualization of software systems and claimed that a easily navigable, interactive and well-constructed 3D visualization is helpful for program comprehension. This 3D visualization can bring the feeling of being "at home" in a software system. Parnin *et al.* [Parnin and Gorg, 2006] proposed an approach to get the context from the developers' interactions with the integrated development environment. Using this context, they demonstrated how to improve the ability of a programmer to recover the mental state associated with tasks and to facilitate the exploration of software through recommendation systems.

Ko *et al.* [Ko *et al.*, 2005] performed a study to investigate the professional Java developers who used integrated development environment Eclipse. Their findings revealed some trends in process of development, as well as some opportunities and challenges for new tools that could save developers up to 35% of the time they currently spend on maintenance tasks. LaToza *et al.* [LaToza *et al.*, 2006] conducted two surveys and eleven interviews to understand developers' activities, practices and typical tools and their satisfaction with each. They found that many problems arose because developers were forced to invest great effort recovering implicit knowledge by exploring code and interrupting teammates and this knowledge was only saved

in their memory. Zhou *et al.* [Zhou and Mockus, 2011] investigated three open source projects to uncover the role of the project environment via the sociality measure at the time a developer joins, affects her future participation.

There are some studies investigated the questions programmers ask during software development, maintenance and evolution. Sillito *et al.* [Sillito *et al.*, 2006, Sillito *et al.*, 2008] conducted two qualitative studies to understand what information a programmer needs to know about a code base while performing a change task and also on how they go about discovering that information. LaToza *et al.* [LaToza and Myers, 2010] performed three studies on reachability questions. Their findings suggested that answering reachability questions is an important source of difficulty understanding large, complex codebases. Begel *et al.* [Begel and Zimmermann, 2014] conducted two survey related to data science applied to software engineering, and grouped a list of 145 questions into 12 categories. Their findings of categorization can help researchers more easily focus their efforts on topics that they are concerned.

Developers always seek and use online resources during software development. Finding the desired knowledge fast and accurately will save plenty of time for developers. Some studies have investigated developers' information needs in software engineering.

Andrew *et al.* [Ko *et al.*, 2006] conducted a study to investigate how developers seek, relate and collect relevant information during software maintenance tasks. The study found that developers spend 35% of their work time in navigation within and between code source files. Andrew *et al.* [Ko *et al.*, 2007] performed a study about developers' day-to-day information needs at a software company. They analyzed the sources developers used, the information that developers sought. In this study, they identified 21 information need types in collocated software development teams.

Lawrance *et al.* [Lawrance *et al.*, 2008] presented a model and algorithm to investigate programmer navigation during software development tasks, named PFIS (Programm Flow by Information Scent). They conducted a study of Java developers, and found the model aggregating human decisions was significantly better than individual developers' decisions. Lawrance *et al.* [Lawrance *et al.*, 2013] investigated how developers debug and revisited from the perspective of information foraging in

a real-world software project. This study treated the developers as predators. They also proposed a more accurate model to predict developer navigation behavior taking into account information scent. Fleming *et al.* [Fleming et al., 2013] investigated three areas: debugging, refactoring, and reuse from the perspective of Information Foraging Theory (IFT). Their findings facilitated better understanding recurring design patterns and developers' information needs.

Buse *et al.* [Buse and Zimmermann, 2012] conducted a survey of 110 professional developers to investigate developers' information needs for artifacts and developers' decision making process. Based on the finding of the survey, they provided some guidelines for analytics tools in software development.

Parnin *et al.* [Parnin and Rugaber, 2012] argued that recovering from an interruption requires extensive time and effort. They presented a conceptual framework to investigate human memory. The empirical evaluation suggested development tools that must satisfy developer information needs and memory aid tools that should be provided in programming task.

Existing studies help to understand what types of information the developers need, where they find this information, and what may facilitate them or prevent them from finding the information they need. These studies were mainly based on qualitative research methods, such as interviews with developers, developers' survey, and observation of developers' task videos. They provide little insights into how to automatically collect and analyze developers' micro-level behavioral data. On the contrary, our study will investigate the micro-level information needs of developers during software development.

2.2 Role of The Web in Software Development

Nowadays, it is common activity that developers search the Web to find information in order to solve the problems they encounter while working on software development task [Brandt et al., 2010, Zagalsky et al., 2012, Nasehi et al., 2012]. Thus, the Web has become an important tool for overcoming programming barriers.

Perpich *et al.* [Perpich et al., 1997] investigated code inspections using the Web to remove inspection bottlenecks. Stylos and Myers [Stylos and Myers, 2006] observed

that software developers often use web search to overcome their learning barrier. They developed Mica, a prototype tool to assist developers more efficiently and effectively use web search to learn how to use APIs. Their work was motivated by the observations of developers using Web resources. Hoffmann *et al.* [Hoffmann et al., 2007] found that general search engines cannot meet the needs of developers which dispersed on many pages: documentation pages, forums and tutorials. They developed a Web search interface, named Assieme to support programming search tasks by combining information including software documentation, sample code and Java Archive (JAR) files.

Because of the prevalence of the Web, some studies focused on recovering traceability links between software artifacts. Dagenais and Robillard [Dagenais and Robillard, 2012] developed a technique to identify code-like terms in software documentation, and link these terms to specific code elements in an API. Jiang *et al.* [Jiang et al., 2017] offered the evidence that developers heavily rely on online API tutorials to facilitate software development. They developed an unsupervised approach to discover relevant web tutorial fragments for APIs, by applying both topic model and PageRank algorithm.

In addition, Stack Overflow has become popular destination where developers exchange knowledge related to computer programming. The Stack Overflow data has attracted much research interest [Anderson et al., 2012, Barua et al., 2014] in recent years. Some tools were developed for assisting software development. Bacchelli *et al.* [Bacchelli et al., 2012] developed Seahawk, an Eclipse plugin that can automatically integrate crowd knowledge of Stack Overflow into Integrated Development Environment (IDE). This tool brings the convenience for developers that they can directly access Stack Overflow data without switching their work context. Ponzanelli *et al.* [Ponzanelli et al., 2014a] presented Prompter, a self-confident recommender system that automatically searches and identifies relevant Stack Overflow discussions under the code context in the IDE. San Pedro *et al.* [San Pedro and Karatzoglou, 2014] proposed RankSLDA, recommending question for collaborative Q&A systems based on developers' topics of expertise. Cordeiro *et al.* [Cordeiro et al., 2012] developed a tool, recommending question answering web resources in IDE based on the information of exception stack traces.

In addition, some researchers have contributed their efforts for program comprehension using Stack Overflow data. Ponzanelli *et al.* [Ponzanelli et al., 2013a] reveal that their tool and Stack Overflow data are capable of sometimes coming up with surprising insights that aid a developer both for program comprehension and software development. Treude *et al.* [Treude et al., 2011] investigated how do developers ask and answer questions on the web. Linares-Vásquez *et al.* [Linares-Vásquez et al., 2014] took an empirical study on how do API changes trigger stack overflow discussions. Recently, Nadi *et al.* [Nadi et al., 2016] performed an empirical investigation into the obstacles developers face while using the Java cryptography APIs based on 100 StackOverflow posts, 100 GitHub repositories, and survey input from 48 developers.

2.3 Recommendation Systems in Software Engineering

Recommendation systems specific to software engineering are emerging to assist developers in a wide range of activities. Robillard [Robillard et al., 2010, Robillard et al., 2014] defined a recommendation system for software engineering to be: “... a software application that provides information items estimated to be valuable for a software engineering task in a given context.”

One kind of information item is code example [Holmes et al., 2005, Klemmer et al., 2000]. XSnippet [Sahavechaphan and Claypool, 2006] was a context-sensitive code assistant framework that allows developers to query a sample repository for code snippets that are relevant to the programming task at hand. The evaluation showed that XSnippet had significant potential to assist developers in finding best-fit code snippets. Holmes *et al.* [Holmes and Murphy, 2005] presented an approach for locating relevant code in an example repository. This approach was based on matching the example code repository to the code under development. Bajracharya *et al.* [Bajracharya et al., 2010] developed Structural Semantic Indexing (SSI), a technique to associate words to source code entities based on similarities of API usage. The experimental evaluation demonstrated that SSI was effective in improving the retrieval of examples in code repositories.

Bacchelli *et al.* [Bacchelli et al., 2012] presented Seahawk, an Eclipse plugin to integrate Stack Overflow crowd knowledge in the IDE. Seahawk allows developers to

find solutions to their programming issues via accessing Stack Overflow data without switching the context. McMillan *et al.* [McMillan et al., 2011] introduced Portfolio, a code search system that retrieves and visualizes relevant functions and their usages. The evaluation showed that Portfolio can help developers find more relevant functions than Google Code Search and Koders. Zhong *et al.* [Zhong et al., 2009] developed a tool MAPO, to mining API usage patterns automatically. MAPO recommends the mined API usage patterns and associated code snippets to help developers locate useful code snippets. The empirical study demonstrated that developers produced code with fewer bugs when facing complex API usages using MAPO. Galenson *et al.* [Galenson et al., 2014] presented CodeHint, a novel technique of interactive and dynamic synthesis of code snippets. It can synthesize real-world Java code that involves native calls, reflection and other language features at runtime.

Another kind of information item is reusable source code. Jacobellis *et al.* [Jacobellis et al., 2014] developed Cookbook, a novel code completion technique. *Cookbook* allows developers to define custom edit recipe by specifying change examples. It can generate an abstract edit recipe that describes the most specific generalization of the demonstrated example program transformations. The evaluation of *Cookbook* demonstrated that it is potential to minimize developers' errors and to speed up manual editing.

Nguyen *et al.* [Nguyen et al., 2012] introduced a context-sensitive code completion tool, called GraPacc. *GraPacc* extracted the context-sensitive features from the code under editing, then ranked the relevant API usage patterns and completed the current code using the chosen pattern. Mooty *et al.* [Mooty et al., 2010] presented a novel Eclipse plugin, called Calcite which instantiates a given class or interface by using code examples. Calcite used crowdsourcing to add to the minuscule instructions in the form of comments which help the developers perform functions that people have identified as missing from the API. The evaluation showed that Calcite improved developers' success rate by 40%. Pletcher *et al.* [Pletcher and Hou, 2009] presented Better Code Completion BCC, an enhancing code completion that developers can use to control how specific API elements should be filtered, grouped and sorted. Foster *et al.* [Foster et al., 2012] developed WitchDoctor, a system that can detect, on the fly,

when a developer is hand-coding a refactoring. The system can then complete the refactoring in the background and propose it to the user long before the user can complete it.

Last but not least, social content in software engineering domain, such as Q&A discussions on Stack Overflow, has recently gained much research interest [Wang et al., 2013, Anderson et al., 2012]. Some work focuses on content recommendation in Q&A sites. For example, Pedro *et al.* [San Pedro and Karatzoglou, 2014] proposed Rank-SLDA, recommending question for collaborative Q&A systems. Wang *et al.* [Wang et al., 2014] presented a tag recommendation system to improve the quality of tags in software information sites. Stack Overflow can also recommend related posts based on topic similarity [Anderson et al., 2012]. Some work attempts to integrate social content into software development environments. For example, Ponzanelli *et al.* [Ponzanelli et al., 2014b] present Prompter, a self-confident recommender system, that automatically searches and identifies relevant Stack Overflow discussions, given the code context in IDE. Zagalsky *et al.* [Zagalsky et al., 2012] present a code search and recommendation tool which brings together social media and code recommendation systems. Others attempt to link information across different information sources. For example, Subramanian *et al.* [Subramanian et al., 2014] present a live API documentation tool, to link the official API documentation with user-generated content on Stack Overflow. Bagheri *et al.* [Bagheri and Ensan, 2014] propose a semantic linking technique to recommend content from Reddit for Stack Overflow questions.

2.4 Summary

This chapter has summarized work on information needs of developers, role of the Web in software development and recommendation systems in software engineering.

Compared with the line of existing work on information needs of developers, our work focuses on micro-level studies of understanding information of developers through web usage.

Compared with the line of existing work on role of the Web in software development, our work is to bridge the gap between developers' intent in natural language queries and content of official software documentations harnessing Web Q&A data.

Compared with the line of existing work on recommendation systems in software engineering, our work is to answer programming questions through software documentation and push undocumented knowledge in software documentation.

Empirical Study: Understanding Information Needs of Developers Through Web Usage

The purpose of computing is insight, not numbers.

— Richard Hamming

Web resources have become key information resources for software development. This chapter reports a micro-level quantitative analysis of the developers' information behavior in seeking and using web resources in software development tasks. The analysis was based on the 29 hours of screen-captured task videos of the 20 developers in two software development tasks. We developed a video scraping tool to automatically extract developers' behavioral data from the task videos, including application usage (*e.g.*, Eclipse, web browser) and application data (*e.g.*, search query, URL, code, exception message). This data allows us to drill into the details of developers' information behavior at a micro level, including patterns of keyword sources, keyword refinement, web pages visited, context switching, and information flow. Our findings reveal three key insights: (1) Need to find more when you know less: developers might have an incomplete or even incorrect understanding of their needs; (2) There is a gap between the producers and consumers of software documentation; (3) Many important pieces of information that developers need are explicitly undocumented in software documentation.

3.1 Background and Motivation

Several studies have investigated the information perspective of software engineering in terms of the ways in which developers seek and use information. Some studies have investigated information needs in software evolution tasks [Sillito et al., 2008] and information needs in collocated development teams [Ko et al., 2007]. Others have explained the developers' information behavior using information foraging

theory [Lawrance et al., 2013, Fleming et al., 2013]. These studies have been focused on developers' information needs within a software project, such as program elements, their change or bug history, and their dependencies.

Open source projects (*e.g.*, github), online technical tutorials (*e.g.*, Java Tutorials, codeproject.com), and technical Q&A platforms (*e.g.*, StackOverflow) contribute a fast-growing body of online knowledge for software development, which software developers have perceived as their "key information resource". Btandt et al. [Brandt et al., 2010] and Li *et al.* [Li et al., 2013] have investigated what online web resources developers seek, and when and how they seek and use web resources. These studies suggest that the ability to seek and use web resources is one of the key abilities affecting the developers' efficiency and success.

Studies on the developers' information behavior often analyze the task videos recorded using screen-capture software during the task. Such task videos provide rich information about the progression of the task, the application usage, and the data accessed and used by the developers. However, performing a micro-level quantitative analysis of such task videos is prohibitively expensive, as this requires transcribing and coding unstructured video data. Studies on human behavior research report that the ratio of recoding time and analysis time can be 1:15-100 [Sanderson and Fisher, 1994], depending on the details and granularity of the information to be collected.

For example, previous study on the developers' online search in software maintenance tasks presented a high-level process-oriented analysis of the developers' online search behavior based on the qualitative observation of the task videos collected in field study [Li et al., 2013]. However, that study provides no insights in the developers' micro-level information behavior and needs in online search, for example, where are keywords from? How often did the developers refine existing queries and /or create new queries? Which web sites were frequently visited by developers? How often did the developers switch their working context? Were there latent types of search sessions? Understanding these information behavior and needs is crucial for designing tools to monitor, diagnose, and react to the developers' web use during software development at a micro level.

In this chapter, we report a micro-level quantitative analysis of the 29-hours task videos collected in our previous study [Li et al., 2013]. We developed a video scraping

tool for automatically transcribing and coding screen-captured task videos. This video scraping tool uses computer vision techniques to recognize window-based applications in screen images, and to extract application-specific data (*e.g.*, code fragments in file editor, error messages in console output, URLs in URL bar, keywords in search box) from the recognized application windows. We have investigated search queries, web pages visited and the patterns of seeking-using web information during software development.

Our study results reveal that: (1) Need to find more when your know less: developers might have an incomplete or even incorrect understanding of their needs; (2) There is a gap between the producers and consumers of software documentation; (3) Many important pieces of information that developers need are explicitly undocumented in software documentation.

3.2 Data Collection

This study is based on the 29-hours task videos of the 20 developers collected in our previous field study of the developers' online search behavior [Li et al., 2013]. In this section, we briefly summarize experimental design of our previous study. We then describe the data extracted from the task videos.

3.2.1 Experimental Design

The study described in [Li et al., 2013] involved two software development tasks. The first task is to develop a new P2P chat software. This task requires the knowledge about multi-threads, socket APIs, and GUI framework (*e.g.*, Eclipse SWT or Java Swing). The second task is to maintain an existing Eclipse editor plugin. This task includes two subtasks. The first subtask is to fix two bugs in the existing implementation. To fix these two bugs, developers need knowledge about Eclipse editor API, Java *IllegalArgumentException*, and plugin configuration. The second subtask asks developers to extend existing editor plugin with file open/save/close features and file content statistics (*e.g.*, word count). This subtask requires developers to use proper Eclipse interfaces and extension points (*e.g.*, *EditorPart*).

We recruited 11 graduate students in the first task, and 13 different graduate students in the second task from the School of Computer Science, Fudan University. The participants were asked to work on the development task individually in a 2-hour session. Participants can search online resources to obtain the knowledge necessary to complete the task. We informed the participants that the goal of the study was not to deliver a working software, but to elicit their genuine information behavior in seeking and using online resources in software development tasks.

The participants used their own computers in the development tasks. All the participants used Windows 7 or newer operating systems. Their development environments are Eclipse3.6 (or newer) or MyEclipse8.0 (or newer). The participants used Google Chrome, Mozilla Firefox, or Internet Explorer to browse the internet. The participants used Microsoft Word to read the task descriptions.

The participants were instructed to use a screen-capture software to record their task process once they started working on the assigned task. These screen-captured task videos are the primary input for this study.

3.2.2 Video Scraping

The task videos of 3 developers in the first task and the task video of 1 developer in the second task were corrupted. As such, this study analyzed the task videos of 8 developers in the first task and the task videos of 12 developers in the second task. In the first task the developers spent on average 94.43 ± 16.21 (mean $\pm$ standard deviation) minutes on the task. In the second task the developers spent on average 83.72 ± 25.07 minutes on the task.

To reduce tedious and time-consuming manual effort, we developed a video scraping tool (*scrRipper*) to collect data from screen-captured task videos in our human studies. The tool assumes that window-based applications are composed of rows and columns of GUI widgets. It has been implemented using *OpenCV*, the popular open source computer vision library. The tool requires only the definition of application window layout and the samples of distinct visual elements (*e.g.*, menu items, toolbar icons) to recognize application windows (*e.g.*, Google Chrome) and their GUI components (*e.g.*, URL bar of Chrome). The tool is insensitive to screen resolution, color

schema, and window theme. It is robust to handle overlapping, stacked, or side-by-side application windows.

The video scraping tool first converts a screen-captured video into a sequence of screen images. It then uses image differencing algorithm to filter out subsequent screen images with no or minor differences (for example due to mouse movement or small scrolling). This produces a sequence of distinct subsequent screen images, $si_1, si_2, \dots, si_{n-1}, si_n$ where any two consecutive images si_i and si_{i+1} are different enough, i.e., over a user-specified threshold (in this study 0.6). Note that the two non-consecutive images can still be the same in this sequence. The tool further identifies screen images with unique content, i.e., distinct content images.

Next, the tool processes one distinct content image at a time. It applies computer vision techniques to detect boundaries, corners and distinct visual elements in the image. It recognizes application windows and their GUI components in the image by composing the detected boundaries, corners and visual elements based on the definition of application window layout. Finally, it scraps user-specified content (e.g., URL bar of Chrome) from the image and uses Optical Character Recognition (OCR) tool (ABBYY FineReader in this study) to convert the scrapped image into text. Figure 3.1 illustrates the input and output of the *scvRipper* tool in this setting. The screencasting software will record the developer's working process as a sequence of screenshots as shown in the upper part of Figure 3.1. Given this task video, the *scvRipper* tool can automatically extract the time-series HCI data from the video as shown in the lower part of Figure 3.1.

3.3 Micro-level Study of Developers' Information Needs

Based on the dataset scraped from the task videos, we performed micro-level quantitative analysis of the information behavior of the 20 developers in the two experimental tasks, focusing on search queries, web pages visited, and patterns of seeking and using information.

Figure 3.1: Extracting time-series HCI data from screen-captured videos by *scvRipper*

3.3.1 Analysis of Search Queries

Search query is an important source for understanding developers' information needs. In this section, we present a fine-grained analysis of search queries developers issued in the two tasks.

(1) Where were the search keywords from?

Given a search query, we determined the sources of its keywords by searching code fragments and console outputs extracted from the distinct Eclipse IDE images before the screen image in which a keyword was used for the first time. If a keyword appears in code fragments, for example, the keyword "openEditor" in the query "java.lang.IllegalArgumentException openEditor" is an Eclipse API used in the source code, we considered its source as "FromCode". If a keyword appears in console outputs, for example, the keyword "IllegalArgumentException" in the above query is an exception thrown in the console view, we considered its source as "FromConsole". If a keyword appears in both code fragments and console outputs, we consider the keyword as "FromCode". If a keyword appears in neither code fragments nor console outputs, we considered its source as self-phrasing, for example, the keywords "eclipse" and "rcp" in the query "eclipse rcp EditorPart EditorInput".

Table 3.1 summarizes the number of distinct keywords the developers used in the two tasks and the sources of these keywords. In the first task the developers' key-

Table 3.1: Statistics of distinct keywords and keyword sources

Developer ID	#DistinctKW	#FromCode	#FromConsole	#Self-phrasing
Taks1				
D1	2	1	0	1
D2	5	0	0	5
D3	13	2	2	9
D4	12	0	0	12
D5	5	1	0	4
D6	13	0	0	13
D7	9	2	1	6
D8	13	1	0	12
mean±standard deviation(D1-D8)	9±4.15	0.87±0.78	0.37±0.69	7.75±4.11
Task2				
D9	10	2	0	8
D10	2	1	0	1
D11	32	14	6	12
D12	10	3	1	6
D13	9	1	0	8
D14	16	5	1	10
D15	13	1	5	7
D16	20	3	3	14
D17	18	5	0	13
D18	18	7	0	11
D19	9	5	0	4
D20	15	6		9
mean±standard deviation(D9-D20)	14.33±7.5	4.41±3.49	1.33±2.05	8.58±3.61

words were mainly self-phrased. In the second task the keywords were both self-phrased and from IDE context.

Table 3.2 presents the top 7 most-used keywords by at least two developers in the first task and the top 11 most-used keywords by at least four developers in the second task. In the first task, all the seven most-used keywords were considered as self-phrasing. 3 of these 7 keywords were from task descriptions (socket, TCP, chat) and 4 described programming language and techniques to be used (Java, SWT, button, event). Using these keywords the developers can find good online examples to complete the first task. They occasionally searched for unfamiliar APIs or errors (e.g., *IProgressMonitor* and *ConnectException*) while modifying reused code examples.

Table 3.2: Most-used keywords in the two tasks

Keywords	Frequency (times)	Who Used These Keywords
Task1		
java	7	D1, D2, D3, D5, D6, D7, D8
socket	5	D1, D3, D4, D5, D7
TCP	4	D2, D3, D6, D8
SWT	3	D5, D6, D8
button	2	D3, D8
event	2	D5, D8
chat	2	D3, D6
Task2		
eclipse	10	D9, D11, D12, D13, D14, D15, D16, D17, D18, D20
plugin	8	D12, D13, D14, D15, D16, D17, D18, D20
EditorPart	6	D10, D11, D12, D17, D18, D19
openEditor	6	D12, D15, D16, D17, D18, D20
IEditorInput	5	D11, D13, D14, D19, D20
doSave	4	D11, D12, D18, D19
editor	4	D11, D12, D14, D16
IWorkbenchPage	4	D14, D17, D18, D20
SWT	4	D9, D11, D16, D17
savefile	4	D12, D13, D17, D20
view	4	D9, D11, D17, D20

In the second task, 6 out of the 11 most-used keywords were considered as self-phrasing, three of which described application platform and techniques to be used (Eclipse, plugin, SWT) and three were from task description (editor, view, savefile). The other 5 most-used keywords were from IDE context, which described Eclipse APIs required for the task (*EditorPart*, *openEditor*, *IEditorInput*, *doSave*, *IWorkbenchPage*). In the second task, developers had to fix bugs of using specific Eclipse APIs and extend specific Eclipse interface. However, using only specific Eclipse APIs often cannot find good online examples to accomplish the second task. Developers had to use application and task context to restrict the search.

(2) How often did the developers refine existing queries and /or create new queries?

Figure 3.2 shows the number of new queries and the number of query refinements in the two tasks. In the first task the developers issued in total 27 new queries (on average 3.37 ± 1.34 new queries per developer). The developers found satisfactory information in the search results of 7 of these 27 new queries and thus did not refine

Figure 3.2: The statistics of new queries and query refinements

the queries. They refined 12 new queries 1-3 times, and 2 new queries more than 3 times. The rest 6 queries were too different from their preceding queries. In our analysis we considered these 6 queries as new queries instead of the refinements of preceding queries. In the second task the developers issued in total 57 new queries (on average 3.37 ± 2.18 new queries per developer). 9 of these 57 new queries were not refined because satisfactory information were found in the search results. 30 new queries were refined 1-3 times, and 9 new queries were refined more than 3 times. The rest 4 queries were considered as new queries because they were too different from their preceding queries.

3.3.2 Analysis of Web Pages Visited

The web pages developers visited directly convey information needs of developers. Thus, we conduct a micro-level analysis about these web pages.

(1) Which web sites were frequently visited by developers?

Our tool can extract web sites from the scraped URLs. We categorized the web sites that the developers visited during the two tasks into seven web categories: search engines (SE), technical tutorials (TT), document sharing sites (DS), topic forums (TF), code hosting sites (CH), Q&A sites (QA), and API specifications (API). Table 3.3 lists the top three most visited web sites of these seven categories in our study.

Table 3.3: The top three most-visited Web sites of 7 Web categories

	The Top 3 Most-Visited Web Sites
Search engines (SE)	www.google.com www.baidu.com www.bing.com
Document sharing sites (DS)	www.360doc.com www.doc88.com www.docin.com
Technical tutorials (TT)	blog.csdn.net www.newasp.cn developer.51cto.com
Topic forums (TF)	topic.csdn.net java.chinaitlab.com www.newsmth.net
Code hosting sites (CH)	download.csdn.net code.google.com github.com
Q&A sites (QA)	zhidao.baidu.com stackoverflow.com iask.sina.com.cn
API specification (API)	docs.oracle.com developers.google.com www.aspose.com

(2) How many web pages were opened after a search?

Figure 3.3 presents the times that the developers opened a specific number of web pages after a search in the two tasks. In the first task the developers did not open any web pages from the search results of 12.5% of the searches. They opened 1-5 web pages for about 77.5% of the searches, and opened 6 or more web pages for 10% of the searches. In the second task the developers did not open any web pages from the search results of 7.14% of the searches. They opened 1-5 web pages for about 80.95% of the searches, and opened 6 or more web pages for 11.9% of the searches.

(3) How often did the developers switch from one web page to another?

Figure 3.4 shows the number of unique URLs (i.e., web pages) that the developers visited in the two tasks and the number of switchings between these web pages. In the first task 5 developers visited less than 9 web pages and made less than 9 web-page switchings. However, the other 3 developers visited on average 20 ± 4 web pages and

Figure 3.3: The statistics of opening a specific number of Web pages

made on average 37.6 ± 19.5 times web-page switchings. In the second task only 2 developers visited less than 6 web pages and made less than 5 web-page switchings. The other 10 developers visited on average 31.4 ± 13.1 web pages and made on average 75.7 ± 38.1 times web-page switchings.

3.3.3 Patterns of Seeking-using Web Information

In this study, we assumed that web browsers were used to seek online information, and Eclipse IDE was used to use online information for accomplishing software development tasks. Based on application usage during the tasks, we studied patterns of seeking and using web information.

(1) How often did the developers switch their working context?

Figure 3.5 shows the number of IDE $\rightleftharpoons$ Browser switchings, Within-Browser switchings, and Within-IDE switchings that the developers performed in every 10 minutes in the two tasks. The box plots label data with 5 attributes. The bottom and top of the box are the first (25%) and third (75%) quartiles (Q_1 and Q_3) of the switchings that the developers performed in a 10-minute time slot. The band inside the box is the second quartile (Q_2 , i.e., the median). The gray boxes indicate the interquartile range

Figure 3.4: Statistics of unique URLs and URL switchings

($IQR = Q3 - Q1$). The lowest end of the whiskers represents minimal observation, and the highest end of whiskers represents maximal observation. The line shows the mean values of the number of switchings over time.

Overall, Within-Browser switchings and Within-IDE switchings occurred more frequently than IDE $\Rightarrow$ Browser switchings in the two tasks. In the first task the developers on average switched 31.87 ± 27.66 times between IDE and web browser, switched 156.37 ± 143.76 times between distinct web contents, switched 72.37 ± 48.15 times between distinct IDE contents. In the second task the developers on average switched 113.12 ± 50.22 times between IDE and web browser, switched 453.75 ± 311.11 times between distinct web contents, and switched 114.08 ± 52.05 times between distinct IDE contents,.

The developers spent on average 64.15 ± 42.84 seconds on a distinct web content in the first task, and spent on average 22.78 ± 12.27 seconds on a distinct web content in the second task. This suggests that the developers had dynamic online search context in which they had to visit a large amount of web contents in a short period of time. The developers spent on average 71.97 ± 57.45 seconds on a distinct IDE content in the first task, and spent on average 44.85 ± 32.67 seconds on a distinct IDE content in the second task. This suggests that the developers' working context in the IDE was dynamic, but was relative stable, compared with their online search context.

Figure 3.5: Statistics of context switchings in every 10 minutes

We studied the relevance of the web contents and the IDE contents surrounding the IDE $\rightleftharpoons$ Browser switchings, Within-Browser switchings, and Within-IDE switchings. We identified only a small number of explicit content copy-paste (64 times in the IDE $\rightleftharpoons$ Browser switchings, 31 times in the Within-Browser switchings, and 42 times in the Within-IDE switchings). This means that information flow is implicit during most of the context switchings.

Context switching causes the loss of working memory. The developers often needed to switch back to the information source to refresh their memory and then go back to use the information. For example, the developer D8 performed 38 times of IDE $\rightarrow$ Browser_{<1min} $\rightarrow$ IDE switchings, which means he frequently switched from the IDE to the web browser and switch back to the IDE within 1 minute. Similarly, the developer D10 performed 20 times such switchings, D12 performed 53 times, D19 performed 67 times.

To further study the implicit information flow within web browser and between IDE and web browser, we built Markov Models [Whittaker and Poore, 1993] for describing the developers' information flow behavior in Within-Browser switchings and

Table 3.4: Markov transition matrices in task 1

		Destination States							
		Eclipse	SE	TT	DS	TF	CH	QA	API
Source States	Eclipse	0	0.16	0.62	0.03	0.11	0	0.07	0.01
	SE	0.27	0	0.34	0.08	0.20	0.05	0.06	0
	TT	0.73	0.20	0	0	0.06	0	0.01	0
	DS	0.38	0.50	0	0	0.13	0	0	0
	TF	0.38	0.45	0.07	0	0	0	0.07	0.03
	CH	0.33	0.67	0	0	0	0	0	0
	QA	0.42	0.42	0.08	0	0.08	0	0	0
	API	0.50	0.50	0	0	0	0	0	0

Table 3.5: Markov transition matrices in task 2

		Destination States							
		Eclipse	SE	TT	DS	TF	CH	QA	API
Source States	Eclipse	0	0.12	0.26	0.28	0.15	0.02	0.01	0.15
	SE	0.19	0	0.26	0.06	0.14	0.03	0.05	0.28
	TT	0.58	0.25	0	0.03	0.05	0	0.02	0.07
	DS	0.81	0.09	0.04	0	0.02	0.02	0	0.02
	TF	0.56	0.25	0.08	0.02	0	0.01	0.01	0.08
	CH	0.45	0.31	0.03	0.07	0.03	0	0.07	0.03
	QA	0.20	0.33	0.10	0	0.10	0.13	0	0.13
	API	0.53	0.27	0.10	0.02	0.02	0.01	0.04	0

in IDE $\Rightarrow$ Browser switchings. The Markov Models consists of 8 states: the 7 web categories (see Table 3.3) and the Eclipse IDE. A transition between the two states represent the switching between the two web categories or the switching between a web category and the Eclipse IDE. The probability of a transition is computed based on the frequencies of the corresponding switchings, i.e., the number of switchings from one state to another state divided by the number of switchings from this state to all the states. Table 3.4 and 3.5 present the the transition probabilities of the Markov Model. The maximal probability of each row is highlighted in bold font.

The table shows that the developers had the highest probabilities to switch between the Eclipse IDE and the technical tutorials in the first task. The technical tutorials seem to be the most useful information source in the first task. In fact, the technical tutorials often contain downloadable code examples that the developers can directly reuse to complete the task. In addition, the developer also integrated the information

found on Q&A sites and API specification sites in the IDE, as indicated by the high probabilities to switch from the Q&A or API specification categories to the Eclipse. In the first task, other than technical tutorials, the developers had the highest probabilities to switch from different web categories (document sharing, topic forum, code hosting, Q&A, and API specification) to the search engine. This suggests that the developers may collect hints from different web sites and then use the hints to search the Internet.

The developers in the second task exhibited different information flow behavior. First, the probabilities to switch from the Eclipse IDE to different web categories (i.e., technical tutorials, document sharing sites, topic forums, and API specifications) are more evenly distributed. Furthermore, unlike the first task, the developers had highest probabilities to switch from technical tutorials, document sharing, topic forums, code hosting sites, and API specifications to the IDE, instead of to the search engine. This suggests that the technical tutorials were not the dominant information sources in the second task. The developers need more diverse information from different sources. Furthermore, the developers were more likely to integrate the information found on these information sources, instead of using the information to further their search.

(2) Were there latent types of search sessions?

Based on Figure 3.1, an item identifies which software the developer uses at what time and with which content.

Definition 3.1 (ITEM) *An item is an application with distinct content in the time-series HCI data.*

It has two attributes: application type and a set of application content (can be empty). We denote an item as $Type\langle Content \rangle$. As such, an item is associated with a list of time stamps that record the time of all the occurrence of the item in the time-series HCI data.

To study the developers' online search behavior during software development, we can segment the time-series HCI data into search sessions based on the occurrence of distinct search queries (i.e., semantic markers) over time, as defined below. The search sessions identify the time periods in which developers search for different information.

Figure 3.6: An illustrative example of search sessions

Definition 3.2 (SEARCH SESSION) A search session is a sequence of items between a search engine (SE) BrowserItem with query $Q1$ and the subsequent search engine BrowserItem with a different query $Q2$.

Figure 3.6 shows an example of two search sessions segmented by the three search engine BrowserItems with different queries, i.e., $BI\langle SE, Q1 \rangle$ at t_1 , $BI\langle SE, Q2 \rangle$ at t_{10} , and $BI\langle SE, Q3 \rangle$ at t_{15} .

To characterize search sessions, we can use a feature space based on a set of descriptive statistics of application usage and content usage in search sessions. In the proof-of-concept prototype, we can automatically extract 12 features from search sessions. Among these 12 features, 6 features represent content usage in a search session: NUMITEMS, NUMBROWSERITEMS, NUMWEBCATEGORIES, NUMKEYWORDS, NUMNEWURLS, NUMIDEITEMS. The other 6 features represent application usage and exploration behavior in a search session: NUMIDEBROWSERSWITCHES, DURATION, NUMCATEGORYSWITCHES, NUMWEBPAGESWITCHES, BROWSERDURATION, IDEEDURATION.

The NUMITEMS, NUMBROWSERITEMS and NUMIDEITEMS counts the number of items, BrowserItems and IDEItems in a search session, respectively. The DURATION, BROWSERDURATION and IDEEDURATION are the time duration (minutes) of a search session and the time spent on BrowserItems and IDEItems in a session. The NUMKEYWORDS counts the number of keywords in the query of a search session. The NUMNEWURLS counts the number of non-search-engine URLs in a search session

Figure 3.7: Heat map of the feature values of the 4 search-session clusters

that are not visited before. The rest NUM...SWITCHES features count the number of item switchings as defined below.

Our approach identifies 168 search sessions in the extracted time-series HCI data. It clusters these 168 search sessions using EM algorithm [Dempster et al., 1977] implemented in Weka [Hall et al., 2009]. We first report our modeling and analysis of search session commonalities and differences.

Figure 3.7 shows the heat map of the normalized average feature values of the 12 features across the four clusters. That is, the sum of each row is equal to 1. Larger values were represented by darker colors, while smaller values were represented by brighter colors. Features in gray background are application usage features, while those in white background are content usage features. The heat map of feature values reveals three distinct meta-clusters: *short* (cluster1), *medium* (cluster2), and *long* (cluster3 and cluster4).

The duration of *short* sessions is very short (less than 1 minute). In these short sessions, the developers mainly used web browsers (6.02 ± 5.86 BrowserItems, compared with 0.30 ± 0.54 IDEItems). They spent on average 93% of total session time in the web

browser, but very little time in the IDE. There were a very small number of switchings (0.56 ± 0.99) between IDE and web browser. The developers opened a small number of new URLs (1.62 ± 0.12), and switched a small number of times between different web pages (2.18 ± 0.47) and between different web categories (0.66 ± 0.94).

The duration of *long* sessions ranges from 4.67 minutes to 104.16 minutes. In these long sessions, the participants frequently used both the web browser and the IDE. There were a large number of switchings (17.63 ± 12.70 in Cluster3 and 30.21 ± 35.46 in Cluster4) between IDE and web browser. Compared with short sessions, the developers opened a large number of new URLs (9.12 ± 3.26 in Cluster3 and 7.26 ± 5.92 in Cluster4), switched a large number of times between different web pages (18.13 ± 4.26 in Cluster3 and 12.53 ± 8.44 in Cluster4) and between different web categories (9.44 ± 2.56 in Cluster3 and 3.26 ± 2.38 in Cluster4).

The statistics of feature values of the *medium* sessions fall in between those of *short* sessions and *long* sessions.

An interesting observation is that search sessions of different clusters differ mainly in application usage features. The differences of content usage features (especially NUMWEBCATEGORIES and NUMKEYWORDS) are smaller across search sessions of different clusters.

Our results identify 4 types of search sessions: *refine* (cluster1), *medium select* (cluster2), *long select* (cluster3), and *integrate* (cluster4). *Refine* sessions are characterized by the least diverse transitions between different types of items, and the high probabilities to transit from the IDE or different categories of web sites to search engine. The model shows that the developers were highly likely to visit search engine after visiting topic forms (TF) and technical tutorials (TT). *Select* sessions are characterized by the most diverse transitions between different types of items with similar transition probabilities. That is, the developers explored many types of online resources in *Select* sessions. *Integrate* sessions are characterized by the transitions mainly between the IDE and different categories of web content. That is, the developers were likely to go back to the IDE after visiting certain online resources in *Integrate* sessions.

Different types of search sessions reflect distinct online search behaviors during software development. *Refine* sessions are *short*. The developers refine search based

Figure 3.8: Timeline plot of search-session types

on a quick exploration of the search results of previous search, for example when they find some hints in the search results or feel that the previous search was unsuccessful. *Select* sessions are *medium* or *long*. The developers explore, compare and select online resources to determine useful information in these search sessions. This exploration and selection process are very diverse and can be time-consuming. *Integrate* sessions are *long*. The developers find useful online resources in these sessions and integrate online resources in the IDE. The integration can take long time.

Figure 3.8 shows the timeline plot of the four types of search sessions in the working process of the 20 developers. Different colors represent different types of search sessions. Black lines represent the ending of the search sessions. Figure 3.8 shows that the developers in the first task (D1-D8) usually had less search sessions than the developers in the second task (D9-D20). The first-task developers had only *refine*, *medium select* and *integrate* sessions, but no *long select* sessions. The second-task developers had the search sessions of all the four types. The two second-task developers (D9 and D10) had Eclipse plugin development experience. They were able to quickly find and

integrate relevant online resources to complete the task in short time. Thus, their task processes were different from the rest 10 second-task developers.

We attributed these differences in the developers' online search behavior to the differences of the two development tasks. The first task is to develop a new P2P chat software using Java socket. The participants can easily find many online code examples on Java socket programming or even code examples implementing similar features as required by the first task. Most participants can successfully modify online code examples without much need for further search. In contrast, the second task is to fix bugs in an Eclipse editor plugin and extend the plugin with new features. The participants in general lacked the knowledge of the Eclipse APIs involved in the task. Unfortunately, no single online resource can cover all the involved APIs. Furthermore, due to the unfamiliarity with these APIs, the developers often encountered unexpected issues while integrating online resources. Thus, they had to keep searching and learning throughout the task.

3.4 Key Insights

These insights were mainly derived from the micro-level study of developers' information needs in Section 3.3.

3.4.1 Need to Find More When You Know Less: Developers Might Have An Incomplete or Even Incorrect Understanding of Their Needs

Based on our pre-study survey of the participants' programming experience (*e.g.*, lines of code they had developed) and their priori knowledge of specific library or framework required for the task, we rated these participants at three levels, A-level (D9, D10) being the most experienced, B-level (D1-D8, D12-D17) being average, and C-level (D18-D20) being the least experienced.

First, from Table 3.1 (the sources of these keywords), the developers' keywords were mainly self-phrased. The queries developers formulated would have a direct impact on the search engine return pages. From Table 3.2, the most-used keywords

are related concepts. When the developers only know some of them, it is beneficial to help developers explore correlated concepts to extend their knowledge or to satisfy their curiosity.

Second, from Figure 3.2 and Figure 3.4, the B-level and C-level developers on average formulated more queries, used more keywords, performed more refinements of existing queries and opened more web pages. For experienced developers, they had less information needs and were able to formulate more precise queries. On the contrary, for the less experienced developers, they might have an incomplete or even incorrect understanding of their needs. They need repeatedly revise their queries and find more related resources to satisfy their needs. Thus, they spent much time in searching and selecting web resources as shown in Figure 3.8. For example, in order to figure out how to open an editor in Eclipse in the second task, the developer D14 first searched *"eclipse editor"*. Unfortunately, the top-6 search results were not relevant to Eclipse plugin development. He then refined the query to *"eclipse plugin editor"*. After browsing several web pages from the new search results, he changed the query to *"eclipse open editor"*. Subsequently, he refined the query as a question *"eclipse how to open editor"*. Finally, he obtained some hints from the search results and refined query to *"eclipse plugin IEditorInput"*.

Based on these observations, the Chapter 4 will address the problem that how to help developers to exploit the crowdsourced knowledge on the correlation of web resources that are highly recognized by programming community.

3.4.2 There Is A Gap Between The Producers And Consumers of Software Documentation

Figure 3.3 shows the number of web pages opened after a search. First, we observed that a small fraction of API documents were visited after developers issued queries. The developers visited API documents just to check information about functionality, structure and parameters. Most of search engine return pages are not very related to issued queries. Thus, the developers need continually refine search queries (Figure 3.2) and collect hints from different web sites (Within-Browser switchings in Figure 3.5) and then use the hints to search the Internet.

Second, from Table 3.4, there is no switch between state “SE” and “API” in task 1, which means that the content of API documentation cannot satisfy developers’ intent. In task 2 (Table 3.5), although some developers visited API documentation after they issued a query, they still need refine their queries to search more related API documentation.

The above observations suggest that this is a gap that exists between software documentation producers and consumers. Because software documentations are written to effectively capture information about functionality, structure and parameters, but lack insights about usage scenarios and restrictions. The Chapter 5 will address this problem on mismatch between the documentation and questions encountered in specific programming tasks because of different wordings.

3.4.3 Many Important Pieces of Information That Developers Need Are Explicitly Undocumented in Software Documentation

Table 3.4 reveals that the developer also integrated the information found on Q&A sites into the IDE, as indicated by the high probabilities to switch from the Q&A to the Eclipse. In task 2, the developers had highest probabilities to switch from technical tutorials, document sharing, topic forums, code hosting sites to the IDE. The developers need more diverse information from different sources rather than only software documentation.

This observation reveals that many important pieces of information that developers need are explicitly documented in software documentation. The undocumented information includes best practices, API usage constraints, typestates, invariants and so on. Especially, to program to an API, developers need to know not only “how to use the AP”, but also “how not to use the API”. By providing important information about functionality, parameters and usage scenarios of an API, API documentation often does a good job at explaining “how to use an API”, but not “how not to use the API”. Thus, the Chapter 6 will address the problem on extracting crowdsourced API negative caveats to augment the software documentation. Such augmentation would raise developer’s caution to avoid misuse of APIs, or to help them fix errors caused by overlooking API negative caveats.

3.5 Summary

In this chapter, we presented a micro-level quantitative study of the 20 developers' web use behavior in the two software development tasks. This study demonstrated the feasibility of using video scraping technique to automatically extract the developer's micro-level behavioral data which would be very difficult or even impossible to gather manually. Our study analyzed search queries, web pages visited, and patterns of seeking-using web information to understand developers' information needs. Three key insights were derived from the micro-level study: (1) Need to find more when you know less: developers might have an incomplete or even incorrect understanding of their needs; (2) This is a gap between the producers and consumers of software documentation; (3) Many important pieces of information that developers need are explicitly undocumented in software documentation.

Discovering Learning Resources from Web Q&A Discussions

Each problem that I solved became a rule which served afterwards to solve other problems.

— Rene Descartes

Software developers need access to correlated information (*e.g.*, API documentation, Wikipedia pages, Stack Overflow questions and answers) which are often dispersed among different web resources. This study is concerned with the situation where a developer is visiting a web page, but at the same time is willing to explore correlated web resources to extend her knowledge or to satisfy his curiosity. Specifically, we present an item-based collaborative filtering technique, named LinkLive, for automatically recommending a list of correlated web resources for a particular web page. The recommendation is by exploiting hyperlink associations from the crowdsourced knowledge on Stack Overflow. We motivate our research using an exploratory study of hyperlink dissemination patterns on Stack Overflow. We then present our LinkLive technique that uses multiple features, including hyperlink co-occurrences in Q&A discussions, locations (*e.g.*, question, answer, or comment) in which hyperlinks are referenced, and votes for posts/comments in which hyperlinks are referenced. Experiments using 7 years of Stack Overflow data show that, our technique recommends correlated web resources with promising accuracy in an open setting. A user study of 6 participants suggests that practitioners find the recommended web resources useful for web discovery.

4.1 Background and Motivation

Software developers perceive online programming resources as the “key information resource” in their learning and work [Brandt et al., 2009]. The ability to search, understand, and use online programming resources is one of the key abilities affecting

software developers' efficiency and success [Tenopir and King, 2004]. In general, web users' information needs can be categorized at a high level as informational, navigational, and transactional [Broder, 2002]. In informational search, there are many situations in which users know the topics they are looking for, and are willing to explore correlated web resources to extend their knowledge or to satisfy their curiosity [Miliaraki et al., 2015, White and Roth, 2009].

For example, when visiting a web page about Singleton design pattern, a developer may be willing to explore correlated web resources, such as other design patterns (*e.g.*, Abstract Factory pattern), concepts related to Singleton pattern (*e.g.*, double-checked locking and enum-based singleton), or Singleton pattern implementations. As another example, consider a developer who visits the JUnit website. The developer may appreciate recommendations of correlated web resources. Examples include alternatives to JUnit ¹ like TestNG, mocking framework for unit testing in Java like EasyMock ², testing frameworks for web development like Selenium, or code analysis tools like PMD.

Search-based methods often cannot help developers discover *correlated and new* web resources [Cooley et al., 1997], because search engines generally employ keyword matching or rely on certain content similarity of web resources. Correlated web resources, however, may not have similar content, for example, the web pages of Singleton pattern, Abstract Factory pattern, and double-checked locking. Furthermore, when developers have no or little knowledge about the correlated information they may be interested in, it is difficult for them to formulate an effective search query. Particularly, for a developer who just starts learning design patterns, it is unlikely that she knows double-checked locking is a concept related to Singleton pattern. Thus, an automatic technique that recommends correlated web resources when developers visit a particular web page can greatly assist them in discovering correlated web resources. Our study addresses this particular need by exploiting the crowdsourced knowledge on Stack Overflow.

¹<https://en.wikipedia.org/wiki/JUnit>

²<https://en.wikipedia.org/wiki/EasyMock>

Question Title	Singleton Pattern Interview
Question Body	I am recently asked about java related question in an interview with following code, since I am very new to java and barely code in Java so I really have no idea what the following code does.
Answer 1	<p>This is a <u>Singleton Pattern</u>.</p> <p>Here's an example of Lazy Initialization, thread-safe singleton pattern from Wikipedia:</p> <p>...</p> <p>Setting the instance variable to <u>volatile</u> tells Java to read it from memory and to not set it in cache.</p> <p><u>Synchronized statements or methods</u> help with <u>concurrency</u>. Read more about <u>double checked locking</u> which is what happens for a "lazy initialization" singleton.</p>
Comment 1	<u>double-checked locking</u>
Answer 2	<p>Interviewer basically wants to check your knoweldge of Singleton pattern . Can the pattern be broken?. Ans is Yes. Check <u>this</u> or google - when singleton is not a singleton.</p> <p>Best course is to use <u>Enum based Singleton</u> as suggested by Joshua Bloch.</p>
Answer 3	<p>For singleton there are two standards that are being used: <u>Double Checked locking</u>, <u>Enum based singleton pattern</u>.</p> <p>UPDATE: Here is another great article which discusses the <u>double checked locking</u>.</p>

Figure 4.1: An example of web linked resources in Stack Overflow. Hyperlinks are underlined.

The idea of recommending correlated web resources for a particular web page is related to recommendation systems for e-commerce websites. In particular, item-based collaborative filtering approaches attempt to summarize product browsing or transaction history to recommend more products that are related to a particular product [Sarwar et al., 2001, Wang et al., 2006, Linden et al., 2003, Li et al., 2005]. Applied to Stack Overflow, the idea is to exploit the crowdsourced knowledge on the correlation of web resources that are highly recognized by the community.

Figure 4.1 shows an example Q&A on Singleton pattern³. In the discussion, users reference 11 web resources related to Singleton pattern. Nine out of these 11 resources

³<http://stackoverflow.com/questions/15425282/singleton-pattern-interview>

are referenced more than 5 times on Stack Overflow. Among frequently referenced web resources on Stack Overflow, some are referenced hundreds or thousands of times. For example, the first web resource for “Singleton Pattern” is referenced 1204 times and the web resource for “Synchronized statements or methods” is referenced 313 times. More importantly, these frequently referenced web resources are often referenced together in the same discussion threads. We hypothesize that taken in aggregate, Stack Overflow discussions can be mined to recommend community-recognized, correlated web resources.

To validate our hypothesis, we conduct a large-scale exploratory study of 5.5 millions hyperlinks referenced in Stack Overflow, to investigate its hyperlink dissemination patterns. We observe that 1) Stack Overflow discussions contain a large number of hyperlinks, that cover a variety of online programming resources (*e.g.*, official APIs, tutorials, code examples, forum discussions); 2) These hyperlinks are widely referenced in questions, answers and comments; 3) Millions of discussion threads contain two or more hyperlinks; 4) The presence of hyperlinks in posts correlates with the community votes on the posts.

Our exploratory study suggests the potential of Stack Overflow data for recommending correlated web resources. Based on our observations of hyperlink dissemination patterns, we design an item-based collaborative filtering approach for recommending correlated web resources. The recommendation uses as features the location in which hyperlinks are referenced (in question, answer, or comment), the co-occurrence frequency of hyperlinks, and the votes on the posts (or comments) in which hyperlinks are referenced. Given a set of Stack Overflow discussion threads, our approach produces a Hyperlink Associative Network (HAN) to model the community-recognized, correlated web resources. Based on this HAN, our approach recommends the top- k most correlated web resources.

To evaluate our approach, we mine a HAN using 6 years of Stack Overflow data (July 2008 - September 2014) and evaluate the accuracy of the recommendation using 9 months of Stack Overflow data (October 2014 - June 2015) as testing dataset. Our evaluation shows that our technique recommends correlated web resources with satisfactory precision and recall in an open setting. As part of this work, we implement

Table 4.1: Top-20 most referenced domains

Rank	#Citations	Domain	Rank	#Citations	Domain
1	1,916,405	stackoverflow.com	11	140,586	apple.com
2	772,453	microsoft.com	12	137,706	python.org
3	667,334	jsfiddle.net	13	124,171	apache.org
4	567,959	github.com	14	112,318	mozilla.org
5	370,405	wikipedia.org	15	101,733	sourceforge.net
6	341,174	google.com	16	83,880	stackexchange.com
7	264,459	php.net	17	79,181	w3.org
8	240,949	oracle.com	18	75,616	mysql.com
9	177,417	android.com	19	65,701	msdn.com
10	172,606	jquery.com	20	63,170	facebook.com

a proof-of-concept tool LinkLive⁴. A user study with 6 participants suggests that our LinkLive tool can recommend helpful and diverse web resources.

4.2 An Exploratory Study of Hyperlinks in Stack Overflow

Our technique relies on the presence of hyperlinks in Stack Overflow discussions, and the presence of potential correlation patterns among these hyperlinks. Thus, an investigation is warranted if hyperlinks in Stack Overflow are a potential source for discovering correlated web resources. We investigate three research questions:

- What kinds of hyperlinks are referenced? Where are they referenced? (Section 4.2.2)
- How frequent a hyperlink is referenced? How frequent two or more hyperlinks are referenced together in the same discussion thread? (Section 4.2.3)
- Does the presence of hyperlinks in posts correlate with number of votes received? (Section 4.2.4)

⁴The LinkLive tool is implemented as a web browser add-on, based on GreaseMonkey/TamperMonkey technique. It can be downloaded at <http://128.199.241.136:9000/download/>.

4.2.1 Dataset

We use Stack Overflow data dump (July 2008 to September 2014) in this exploratory study. The same dataset is also used as the training data for the evaluation of our technique. The dataset contains 7,990,787 questions, 13,684,117 answers, and 32,506,636 comments. We consider a question and all its answers and comments as a discussion thread. Thus, we have 7,990,787 discussion threads in the dataset, *i.e.*, the same as the number of questions. We extract hyperlinks in questions, answers and comments from the *href* HTML tag. As many hyperlinks in comments are referenced as plain text, we also use regular expressions to parse plain text and to extract hyperlinks.

4.2.2 What and Where About “Referenced”

We extract 5,522,886 distinct hyperlinks from the dataset. These hyperlinks are from 234,815 distinct domains. Table 4.1 lists the top-20 most referenced domains in Stack Overflow discussions. Observe that the domains cover a variety of online programming resources, including official API documents, tutorial websites, code example sites, code repositories, Wikipedia, and forum discussions, etc. Our results are consistent with earlier studies on hyperlink sharing practices on Stack Overflow [Gomez et al., 2013].

Tables 4.2 and 4.3 summarize the statistics of hyperlinks that are referenced in questions, answers, and comments, respectively. We can observe that hyperlinks widely present in questions, answers, and comments. Stack Overflow encourages users including hyperlinks to relevant web resources in their discussions⁵. This explains the wide present of hyperlinks in Stack Overflow. Therefore, to analyze hyperlinks in Stack Overflow, we must consider all components that may contain hyperlinks in the discussion, *e.g.*, questions, answers, and comments.

4.2.3 Hyperlink Reference and Co-occurrence Frequency

Figure 4.2 plots hyperlink citation distribution and domain citation distribution. The chart shows the percentage of hyperlinks (or domains) that have been referenced for a

⁵<http://stackoverflow.com/help/how-to-answer>

Table 4.2: Statistics of questions/answers/comments having hyperlinks

	#Questions	#Answers	#Comments
Total number	7,990,787	13,684,117	32,506,636
Contain hyperlinks	1,315,053 (16.46%)	4,457,576 (32.57%)	5,788,648 (17.81%)

Table 4.3: Statistics of the sources of distinct hyperlinks

Location of hyperlink	Questions	Answers	Comments	All
#Distinct hyperlinks	1,379,007	3,120,816	1,913,474	5,522,886

4.2.a: Hyperlink distribution

4.2.b: Domain distribution

Figure 4.2: The hyperlink and domain citation distribution

certain number of times. About 21.83% hyperlinks and about 50.14% domains are referenced at least twice in discussions. The power-law distribution of hyperlink citations indicates that the hyperlinks shared in Stack Overflow are largely stable, although the number of distinct hyperlinks keeps growing. That is, a small portion of frequently referenced hyperlinks attracts a large portion of developers' attention. Therefore, the frequency of a hyperlink is an important indicator of the community's preference.

Figure 4.3 shows distribution of hyperlinks per discussion thread. Observe that 28.65% of discussion threads (*i.e.*, 2, 289, 360) contain two or more hyperlinks. This shows that millions of discussion threads can be a potential source for mining correlated web resources. Sharing in the same discussion thread indicates the relatedness among hyperlinks.

Figure 4.3: Distribution of number of hyperlinks per discussion thread

4.2.4 Hyperlink-Vote Correlation

Previous studies show that, the presence of hyperlinks in a post is a strong indicator of the post being more informative [Dou et al., 2009]. We want to further investigate the correlation between the presence of hyperlinks in a post and the number of votes it receives. To this end, we collect 4,596,855 accepted answers in our dataset. We then split these accepted answers into two groups: group1 (1,640,651 answers) where the accepted answers contain hyperlinks, and group2 (2,956,204 answers) where the accepted answers do not contain hyperlinks. We assume the null hypothesis H_0 : There is no statistically significant difference in the number of votes on the accepted answers from the two groups.

Examination of distribution of number of votes on answers shows that the distribution does not obey normal distribution. Therefore, we use non-parametric statistical tests to study the significance of the vote differences in the two groups. In particular, we use the KolmogorovSmirnov test (KS-test) [Lilliefors, 1967]. The KS-test has the advantage of making no assumption about the distribution of data. *p-value* of the KS-test result is below 0.001. Therefore, we reject the null hypothesis. In other words, there is a statistically significant difference in the number of votes on the accepted answers with and without hyperlinks. This analysis suggests that number of votes is another important indicator of community's preference of hyperlink.

Table 4.4: The notations of this chapter

Symbol	Description
T	A discussion thread
h_s	A seed ROI
h_t	A candidate ROI
H	The set of distinct hyperlinks $H = h_s \cup h_t$
$\mathcal{G}_{h_s} = (V, E)$	Hyperlink Co-occurrence Graph for h_s , Vertex V , Edge E
$e_{s,t}$	The edge from node h_s to node h_t
$HAN = (H, X)$	Hyperlink Associative Network. $HAN = \bigcup \mathcal{G}_{h_s}$
$\omega_{s,t}$	The weight of edge $e_{s,t}$ in HAN
$S_{e_{s,t}}$	The score of edge $e_{s,t}$ in E
$S'_{e_{s,t}}$	The normalized score of edge $e_{s,t}$ in E
S_p	The score vector of posts (question and answers) in the discussion thread T
S_c	The score vector of comments of a post in the discussion thread T

4.3 The LinkLive Approach

Our exploratory study shows that hyperlinks in Stack Overflow are a good source for mining correlated web resources. The analysis also leads to the design of our item-based collaborative filtering approach for mining correlated web resources. In this work, we consider a hyperlink as a Resource-of-Interest (ROI). Given a hyperlink (referred to as a *seed hyperlink* or *seed ROI*), our goal is to recommend top- k correlated web resources (referred to as *recommended hyperlinks* or *recommended ROIs*) that are highly recognized by the Stack Overflow community.

4.3.1 Hyperlink Associative Network

We construct a Hyperlink Associative Network (HAN) to model correlated hyperlinks in Stack Overflow as follows. The notations used in this chapter are listed in Table 4.4.

Definition 4.3 (Discussion Thread) *A discussion thread consists of a question and all its answers in chronological order. Both questions and answers are also known as posts. A post may have zero or more comments.*

Definition 4.4 (ROI Location and Location Type) *The question, answer, or comment in which a ROI is referenced is the location of the ROI. Each of which is also known as the location type.*

Figure 4.4: Hyperlink co-occurrence graph

Definition 4.5 (ROI Cascade) A ROI cascade consists of all the discussion threads that reference a seed ROI in chronological order.

In our model, a discussion thread is the basic information unit. Figure 4.4 illustrates multiple discussion threads, from Thread1 to ThreadM, in chronological order. Let H be the set of distinct hyperlinks in all discussion threads. For seed ROI $h_s \in H$, we construct a ROI cascade, as shown in Figure 4.4, where the seed ROI is highlighted in red. The seed ROI is referenced in multiple locations including questions q of Thread1 and ThreadM, answer A2 in Thread3, and a comment to answer A4 in Thread4. Note that, Thread2 does not contain the seed ROI, and thus is excluded in the ROI cascade.

Definition 4.6 (Hyperlink Co-occurrence Graph $\mathcal{G}_{h_s}$) A $\mathcal{G}_{h_s} = (V, E)$ is a directed graph where vertex set V is the set of all distinct hyperlinks referenced in ROI cascade of seed ROI h_s , and a co-occurrence edge $e_{s,t} \in E$ represents the co-occurrence of the seed hyperlinks h_s and the other hyperlink h_t in the same discussion thread. h_t is referred to as candidate hyperlink or candidate ROI. The edge $e_{s,t} \in E$ is indexed by the locations of h_s and h_t . The edge has a score indicating the number of votes on the post (i.e., question or answer) or comment in which h_t is referenced.

Given the ROI cascade of seed ROI h_s , the edge set E of graph $\mathcal{G}_{h_s}$ is constructed as follows. For each discussion thread T in the ROI cascade,

- If the location of h_s is a post (i.e., question or answer), then for each distinct hyperlink h_t in the posts of the discussion thread T , a co-occurrence edge $e_{s,t}$ (node h_s to node h_t) is added to E . The red edges in Thread1, Thread3, and ThreadM and the orange edge between the seed ROI and *url6* in question Q of Thread3 illustrate this scenario. The seed ROI h_s and a candidate ROI h_t may appear in the same post, e.g., *url7* in answer A2 of Thread3. If a candidate ROI h_t appears in n ($n \geq 2$) different posts, e.g., *url13* in answers A1 and A4 of ThreadM, then n different edges will be added.
- If the location of h_s is a post, then for each distinct hyperlink h_t in the comments of the post, a co-occurrence edge $e_{s,t}$ (node h_s to node h_t) is added to E . The purple edges between the seed ROI and *url1* and *url2* in comments of question q of Thread1 illustrate this scenario. Note that we do not consider hyperlinks in comments of other posts (such as the one in a comment of A4 of Thread1) as candidate ROIs. The rationale is that the comments are usually related to only the post being commented.
- If the location of h_s is a comment, then for each distinct hyperlink h_t in the comments of the same post and in the post being commented, a co-occurrence edge $e_{s,t}$ (node h_s to node h_t) is added to E . The orange and purple edges between the seed ROI and *url10* in answer A4 of Thread4 and *url11* and *url12* in comments of answer A4 illustrate this scenario. Again, we do not consider hyperlinks in other posts (such those in answers A1 and A2 of Thread4) as candidate ROIs of the seed ROI in a comment of a post.

Definition 4.7 (Hyperlink Associative Network HAN) A $HAN = (H, X)$ is a directed graph where vertex set H is the set of hyperlinks in all discussion threads, and an associative edge $e_{s,t} \in X$ if there exist one or more co-occurrence edges $e_{s,t} \in E$ (E is the edge set of $\mathcal{G}_{h_s}$). That is, $HAN = \bigcup \mathcal{G}_{h_s}$ for all h_s and $H = h_t \cup h_s$.

Each associative edge $e_{s,t} \in X$ has a *weight*, to be described next. This weight measures the correlation similarity between seed ROI h_s and a candidate ROI h_t .

4.3.2 Associative-Edge Weight Computation

The *weight* of associative edge $e_{s,t} \in X$ (denoted by $\omega_{s,t}$) is computed based on the *score* of the corresponding co-occurrence edges $e_{s,t} \in E$ (denoted by $S_{e_{s,t}}$) in $\mathcal{G}_{h_s}$. For a candidate ROI h_t of seed ROI h_s in a discussion thread, the score of posts (or comments) in which the h_t is referenced reflects the competition among the candidate ROIs in the discussion thread. The intuition is that ROIs in high-score posts (or comments) would be more valuable to users than those in low-score posts (or comments). Thus, a naive method to compute $\omega_{s,t}$ is to sum up $S_{e_{s,t}}$ of all co-occurrence edges $e_{s,t} \in E$, i.e., $\omega_{s,t} = \sum_{e_{s,t} \in E} S_{e_{s,t}}$. We refer to this straightforward method as the *baseline method* in our evaluation.

We now design an enhanced weight computation method (referred to as the *enhanced method* in our evaluation) based on the following two intuitions. First, the correlation between a seed ROI h_s and a candidate ROI h_t is different when h_s is referenced in different types of locations, i.e., question, answer, and comment. This intuition is based on the consideration that the asker is very likely to follow up all answers, but an answerer or a commenter may not pay much attention to answers or comments from others. Second, scores of posts or comments vary greatly from one discussion thread to another. Thus, it would be necessary to normalize the scores within each discussion thread so that the scores are comparable across different discussion threads.

Based on above two intuitions, we normalize the *score* of co-occurrence edge $e_{s,t} \in E$ for the three different types of locations of h_s as follows. We denote the normalized score as $S'_{e_{s,t}}$.

h_s is referenced in question. The seed ROI in question Q of Thread1 and ThreadM in Figure 4.4 illustrate this scenario. We only consider the competition among the candidate ROIs h_t in comparable type of location, i.e., h_t in posts or h_t in comments, because the score of posts and comments can vary greatly in scale and most of comments have no score.

Let $\mathbf{S}_p$ be the score vector of all posts (question and answers) of the discussion thread T . If h_t is referenced in a post in T , then the normalized score of the edge $e_{s,t} \in E$ is:

$$S'_{e_{s,t}} = \frac{S_{e_{s,t}} - \min(\mathbf{S}_p)}{\max(\mathbf{S}_p) - \min(\mathbf{S}_p)} \quad (4.1)$$

That is, $S'_{e_{s,t}}$ is a value in $[0, 1]$ that reflects the relative score of h_t compared with that of other candidate ROIs of the h_s in the posts of the discussion thread T . If $\max(\mathbf{S}_p) = \min(\mathbf{S}_p) = 0$, we set $S'_{e_{s,t}} = 0.1$. The *score* of the red edges in Thread1 and ThreadM in Figure 4.4 is normalized using Eq. 4.1.

Let $\mathbf{S}_c$ be the score vector of all comments of question q in which h_s is referenced in discussion thread T . If h_t is referenced in a comment of question q , then the normalized score of edge $S'_{e_{s,t}}$ is:

$$S'_{e_{s,t}} = \frac{S_{e_{s,t}} - \min(\mathbf{S}_c)}{\max(\mathbf{S}_c) - \min(\mathbf{S}_c)} \quad (4.2)$$

If $\max(\mathbf{S}_c) = \min(\mathbf{S}_c) = 0$, we set $S'_{e_{s,t}} = 0.1$. The *score* of the purple edges in Thread1 in Figure 4.4 is normalized using Eq. 4.2.

h_s is referenced in answer. The seed ROI in answer A2 of Thread3 in Figure 4.4 illustrates this scenario. Let the answer be in discussion thread T . If h_t is referenced in question q of T (e.g., the orange edge between the seed ROI and *url6* in question Q of Thread3), we set $S'_{e_{s,t}} = 1$. This is based on the consideration that the information in an answer should be directly related to the information in the question. If h_t is referenced in an answer of discussion thread T (e.g., the red edges between seed ROI and *url7*, *url8* and *url9* in Thread3), we compute $S'_{e_{s,t}}$ using the Eq. 4.1. If h_t is referenced in a comment of an answer in which h_s is referenced, we compute $S'_{e_{s,t}}$ using the Eq. 4.2.

h_s is referenced in comment. The seed ROI in a comment of answer A4 of Thread4 in Figure 4.4 illustrates this scenario. If h_t is referenced in a post (A4 of Thread4) being commented (e.g., the orange between seed ROI and *url19* in answer A4 in Thread4), we set $S'_{e_{s,t}} = 1$. This is based on the consideration that the information in a comment should be directly related to the information in the post being commented. If h_t is referenced in the comments of the same post (e.g., the purple edges between seed ROI and *url11* and *url12* in the comments of answer A4 of Thread4), we compute $S'_{e_{s,t}}$ using Eq. 4.2.

Once we normalize the *score* of co-occurrence edge $e_{s,t} \in E$ for all discussion threads, we compute the normalized weight of associative edge $e_{s,t} \in X$ as $\omega_{s,t} = \sum_{e_{s,t} \in E} S'_{e_{s,t}}$.

4.3.3 ROI Recommendation

The HAN mined from a set of discussion threads serves as the underlying model for recommendation of correlated web resources. Given a ROI, if it appears in the HAN, we recommend the top- k candidate ROIs in the HAN that have the highest associative edge weight (*e.g.*, correlation similarity) with the given ROI as the recommended ROIs.

4.4 The LinkLive Tool

We implement a proof-of-concept tool of our LinkLive approach⁶. The backend hyperlink associative network (*i.e.*, hyperlink correlation model) is mined from training data, which is the Stack Overflow data dump (July 2008 to September 2014) (see Section 4.2). When a user visits a web page or mouse hovers over a hyperlink in the web page, the LinkLive tool searches for hyperlinks from the backend model. If the hyperlink is found in the backend, and the hyperlink has been referenced 5 times or more in the training dataset, the tool recommends top-10 correlated web resources for the given hyperlink. We set a minimal reference frequency of the seed hyperlink for triggering the recommendation, because we observe that for seed hyperlinks that are referenced fewer than 5 times, the co-occurring hyperlinks are ad-hoc.

Figure 4.5 shows the LinkLive recommendation when mouse hovers over hyperlink (https://en.wikipedia.org/wiki/Singleton_pattern) in a Stack Overflow question⁷. In addition to the recommendation of correlated web resources, the LinkLive tool also shows a bar chart of citation history for the seed hyperlink and each recommended hyperlink.

4.5 Evaluation of the LinkLive Recommendation

We evaluate LinkLive from two perspectives: *accuracy* and *reliability*.

⁶ <http://128.199.241.136:9000/download/>

⁷ <http://stackoverflow.com/questions/23360052>

Figure 4.5: The LinkLive tool

- How accurate is the LinkLive recommendation of correlated web resources compared with user-explicitly-referenced web resources in Stack Overflow discussions? (Section 4.5.2)
- How does the time elapsed after modeling training affect the accuracy of the LinkLive recommendation? (Section 4.5.3)

4.5.1 Experiment Setup

Testing Dataset. To answer the above research questions, we use Stack Overflow data dump (October 2014 to June 2015) as the testing dataset. The testing dataset contains 1,835,754 discussion threads and 1,384,063 distinct hyperlinks. Similar to the training dataset, 28.36% of discussion threads (or 520,619) in the test dataset have two or more hyperlinks.

We build the “ground truth” to answer the above research questions from these 28.36% discussion threads as follows. We collect the discussion threads in the testing

dataset that reference at least two hyperlinks and at least one of them can trigger LinkLive recommendation. 260,141 discussion threads in the testing dataset satisfy this criteria, and 92,800 hyperlinks can trigger LinkLive recommendation. These 92,800 hyperlinks are referred to as seed hyperlinks in this evaluation. Given one of the 260,141 discussion threads and a seed hyperlink in the discussion thread, we collect all the co-occurring hyperlinks for the seed hyperlink in the discussion thread in the same way as we build Hyperlink Co-occurrence Graph (see Section 4.3.1). We consider this set of co-occurring hyperlinks as the ground truth of correlated web resources for the given seed hyperlink in the test data. Hereafter, we refer to this ground truth as user-explicitly-referenced correlated web resources.

Metrics. We use two metrics to evaluate the accuracy of LinkLive recommendation: *Precision@k* and *Recall@k* ($Pr@k$ and $Re@k$ for short). $k = \{1, 5, 10, 20, 30\}$ is the number of recommended web resources. Let $R_{h_s}^k$ be the set of top- k recommended web resources for a seed hyperlink h_s using LinkLive. Let GT_{h_s} be the set of user-explicitly-referenced correlated web resources for h_s in a discussion thread. $Pr@k$ for h_s is $\frac{|R_{h_s}^k \cap GT_{h_s}|}{k}$. $Re@k$ for h_s is $\frac{|R_{h_s}^k \cap GT_{h_s}|}{|GT_{h_s}|}$. We average the precision and recall values over all the 260,141 discussion threads.

4.5.2 The Accuracy of LinkLive Recommendation

Recall that we have a baseline method and an enhanced method to compute associative edge weight (the correlation similarity between hyperlinks) for the recommendation of correlated web resources (see Section 4.3.3). In this section, we compare the accuracy of the recommendation of the two methods using the *Precision@K* and *Recall@K* metrics.

Overall Performance. Fig 4.6 reports the $Pr@k$ and $Re@k$ of both baseline and enhanced methods. We observe that:

- The enhanced method outperforms the baseline method at all different k values for both precision and recall. This shows that taking into account the location where hyperlinks are referenced and normalizing scores in discussion threads improves the accuracy of LinkLive recommendation.

Figure 4.6: Overall performance of the LinkLive recommendation

- As expected, precision drops as the value of k increases, while the recall increases along k increases. The best precision is 9.34% at $k = 1$ and the best recall is 15.61% at $k = 30$. At $k = 10$, the precision is 3.03% and the recall is 10.86% .

Note that $Pr@k$ is calculated by $\frac{R_{hs}^k \cap GT_{hs}}{k}$. Most discussion threads only contain two hyperlinks. That is, $R_{hs}^k \cap GT_{hs} = 1$ in most case. Thus, the $Pr@k$ is expected to be very small. On the other hand, in the testing dataset, 79% of hyperlinks have not been referenced in the training dataset.

As discussed in Section 4.2.3, these hyperlinks do not represent web resources that the Stack Overflow community care the most about. However, the presence of such hyperlinks brings down the precision of the recommendation. Furthermore, there are only a small percentage (about 3.81%) of discussion threads referencing 5 or more hyperlinks. That is, for over 96% of the 260,141 discussion threads from which we collect the ground truth, the ground truth contains fewer than 5 user-explicitly-referenced web resources. Therefore, even the LinkLive top- k recommendations contain all the user-explicitly-referenced web resources, the precision is low when k is 5 or larger.

As the main goal of LinkLive recommendation is to help developers discover correlated web resources that they may be interested in, we deem recall to be more important than precision. The reference of hyperlinks in the test discussion threads can be affected by many factors, such as variation of question topics, the expertise of askers and answerers. In such an open-ended setting, our enhanced recommendation

Table 4.5: Number of seed ROIs and citations at different citation levels

Citation Levels	5-50	51-100	101-500	>500
#Seed ROIs	82,190	5,979	4,171	460
#Citations	233,707	48,130	79,396	59,125

method achieves satisfactory and acceptable recall (10.86% at top 10), on a par with the recall of the state-of-the-art recommendation systems for e-commerce or location-based services reported in the literature [Yuan et al., 2014, Grbovic et al., 2015b].

Impact of Number of Citations. As shown in Fig 4.2, the distribution of hyperlinks obeys a power-law distribution. It reveals that most hyperlinks are referenced by a small number of times and a small fraction of hyperlinks are referenced frequently. In this evaluation, we split the hyperlinks into four citation levels according to the number of their citations in Q&A discussions: “5 – 50”, “51 – 100”, “101 – 500” and “> 500”. In the testing dataset, the number of seed hyperlinks at these 4 citation levels are 82,190, 5,979, 4,171 and 460, respectively. These seed hyperlinks are referenced 233,707, 48,130, 79,396 and 59,125 times in the testing dataset, respectively.

Tables 4.6 and 4.7 show the precision and recall of LinkLive recommendation for seed hyperlinks at the 4 citation levels. From these two tables, we observe that:

- At all citation levels, the enhanced method outperforms the baseline method.
- Both precision and recall increase as the citation frequency of seed hyperlinks increases. For seed hyperlinks that are referenced “5 – 50” times, the precision and recall are worse than the overall performance. For seed hyperlinks that are referenced “> 500” times, the enhanced method achieves precision 4.81%@10 and recall 17.61%@10, which is better than the overall performance.
- Comparing the lowest and highest citation levels “5 – 50” versus “> 500”, the performance at citation level “> 500” is significantly better than that of level “5 – 50”. In most cases, the precision and recall values are doubled between the two levels.

Table 4.6: Precision at different citation levels

Citation Levels		5-50	51-100	101-500	>500
Pr@1	Enhanced	7.02%	9.72%	11.46%	15.31%
	Baseline	4.42%	5.24%	7.24%	12.86%
Pr@5	Enhanced	3.31%	4.51%	5.49%	7.05%
	Baseline	2.48%	2.83%	3.49%	5.64%
Pr@10	Enhanced	2.25%	2.79%	3.69%	4.81%
	Baseline	1.86%	1.93%	2.37%	3.47%
Pr@20	Enhanced	1.56%	1.68%	2.23%	2.27%
	Baseline	1.37%	1.40%	1.53%	2.06%
Pr@30	Enhanced	1.23%	1.24%	1.63%	1.98%
	Baseline	1.12%	1.13%	1.19%	1.49%

Table 4.7: Recall at different citation levels

Citation Levels		5-50	51-100	101-500	>500
Re@1	Enhanced	2.81%	3.94%	4.68%	7.25%
	Baseline	1.75%	2.09%	2.80%	5.81%
Re@5	Enhanced	6.09%	8.58%	10.12%	13.69%
	Baseline	4.64%	5.31%	6.27%	10.86%
Re@10	Enhanced	7.93%	10.46%	13.02%	17.61%
	Baseline	6.62%	7.17%	8.42%	12.75%
Re@20	Enhanced	1.39%	12.33%	15.49%	19.87%
	Baseline	9.19%	9.70%	10.81%	14.90%
Re@30	Enhanced	11.77%	13.65%	16.83%	21.22%
	Baseline	10.76%	11.47%	12.50%	16.14%

This result suggests that for a seed hyperlink that is more frequently referenced in Q&A discussions, hyperlinks that previously co-occur with seed hyperlink are very likely to be referenced again, when the seed hyperlink is referenced again. As such, our approach makes more accurate recommendation for the seed hyperlinks that are more frequently referenced. This phenomenon can be explained by using preferential attachment theory [Bretherton, 1985], which is the key intuition underlying the design of our LinkLive approach.

Figure 4.7: Performance on the three testing subsets

The LinkLive recommendation is reasonably accurate in an open-ended setting, compared with user-explicitly-referenced correlated web resources.

4.5.3 The Reliability of LinkLive Recommendation

Stack Overflow receives thousands of questions and answers every day. As the data grows over time, the technology landscape also changes rapidly. LinkLive recommendation relies on hyperlink correlation patterns in Q&A discussions to make effective recommendation. The time elapsed after model training may invalidate the patterns learned from the past Q&A discussions. To study the impact of time elapsed on LinkLive recommendation, we split the testing dataset into three subsets, each of which contains 3-month data (October 2014 to December 2014, January 2015 to March 2015, and April 2015 to June 2015). We use the first 3-month data as the baseline to compare the accuracy of LinkLive recommendation with the second and the third 3-month data. For the three subsets, we collect 54,654, 51,076 and 47,722 seed hyperlinks, respectively, and LinkLive tool recommends 181,385, 169,511 and 158,381 correlated web resources, respectively. For each subset, we collect the ground truth and compute $Precision@k$ and $Recall@k$ in the same way as we process the full testing dataset.

Figure 4.7 shows $Precision@k$ and $Recall@k$ values ($k = 1, 5, 10, 20, 30$) for the three testing subsets. We observe that:

- Both precision and recall deteriorate as the time gap between the training dataset and the testing dataset increases. However, neither precision nor recall degrade significantly even after six months of HAN training.
- The time elapsed after model training has bigger impact on precision than on recall. Precision drops up to 15.4% after three months of model training, up to 20.9% after six months of model training. Recall drops 9.65% after three months of model training, 15.2% after six months of model training.

The LinkLive tool makes reliable recommendation for new data after three months of model training. Recall is more reliable than precision which is a desirable goal of the tool. Users can update the backend model every three months to avoid significant drop in recommendation accuracy.

4.6 User Study

We perform a user study to evaluate the helpfulness and diversity of LinkLive recommendation:

- Can the LinkLive discover helpful and diverse web resources for developers in practice?

4.6.1 Study Design

Participants. We recruit 6 graduate students. All participants have either computer science or computer engineering background. Participants have 1-3 years of programming experience in popular programming languages and tools, such as Java, Python, C++, and MySQL.

Data Sampling. From the testing dataset, we randomly sample 45 seed hyperlinks that are referenced in 45 best answers (one seed hyperlink per answer) for each hyperlink citation level (“5 – 50”, “51 – 100”, “101 – 500” and “> 500”). We collect in total 180 (45x4) seed hyperlinks for this study. As the evaluation of the recommended web resources requires certain background, we sample hyperlinks that are referenced

in the discussion threads that are tagged with programming techniques (*i.e.*, Java, Python, C++, MySQL) that participants are familiar with. Each participant is randomly assigned 30 seed hyperlinks to rate the helpfulness and diversity of recommended web resources by the LinkLive tool.

Evaluation Metrics. For each sampled seed hyperlink, we use the LinkLive tool to recommend the top-10 correlated web resources. We implement a web application for the participants to evaluate the recommended web resources. For each seed hyperlink, the web application presents the seed hyperlink, the answer in which the seed hyperlink is referenced, and the top-10 recommended web resources for the seed hyperlink. Participants are asked to rate each recommended web resources in terms of helpfulness and category.

Helpfulness is a 7-point likert scale (1 being least helpful to 7 being most helpful). Participants are asked to read carefully the information in the seed hyperlink, the answer in which the seed hyperlink is referenced, and the recommended web resources to determine the level of helpfulness of the recommended web resources for understanding the answer and/or the content of the seed hyperlink.

We predefine 5 categories for the web resources, including official documentation (*e.g.*, Java API documentation, jQuery library API), unofficial documentation (*e.g.*, technical blogs, jsfiddle code examples), Q&A site (*e.g.*, Stack Overflow, Quora), code repository (*e.g.*, Github, Sourceforge), and encyclopedia (*e.g.*, Wikipedia, Javapedia). Participants are asked to select one category for each recommended web resource. They can enter “others” if they believe none of the predefined categories fit the recommended web resources, for example dead links.

4.6.2 Perceived Helpfulness

Among the 1,800 recommended web resources (10 for each 180 seed hyperlinks), 71.8% are rated helpful (5=28.7%, 6=24.7%, or 7=18.3%), 12.05% are rated neutral (4), and 13.6% are rated as unhelpful (1=3.1%, 2=3.0% or 3=7.5%). Among the 13.6% unhelpful resources, 2.4% are dead links (*e.g.*, due to changing URL), and most others are download links or home pages of API documentation or library, which offer little information.

Figure 4.8: Average recommendation helpfulness score at different citation levels

For each seed hyperlink, we average the helpfulness score of the 10 recommended web resources. Figure 4.8 shows the box plot of the average recommendation helpfulness score for the 45 sampled seed hyperlinks at different levels of hyperlink citation. Observe that there is no significant difference in the mean perceived helpfulness score at different levels of hyperlink citation. The mean perceived helpfulness score is slightly above 5 (*i.e.*, moderate helpful). There is one outlier seed hyperlink at level “> 500”, *i.e.*, <http://stackoverflow.com/help/on-topic> which describes questions that can be asked on Stack Overflow. For this seed hyperlink, 7 of the 10 recommended resources are about the norms or good practices to ask or answer questions on Stack Overflow. Although the recommend web resources are very relevant to the seed hyperlink, the participant deems the recommendation as unhelpful, as they are not relevant to any specific programming issues.

Figure 4.9 presents the box plot of the average recommendation helpfulness score for the 30 sampled seed hyperlinks for each participant. We can see that Participant1 has the least variation in his/her ratings, while Participant6 has the most variation in his/her ratings. The other four participants have similar variations in their ratings. Five participants have similar mean perceived helpfulness score in their ratings (around 5, moderate helpful), while Participant1 have slight higher mean perceived helpfulness score (around 6). For Participant4, there is one outlier seed hyperlink, *i.e.*, <https://docs.python.org/2/library/struct.html>. The hyperlink links to the Python module for conversions between Python values and C structs. Participant4 gives the score of 2 to 4 recommended web resources (about audio, abstract syntax

Figure 4.9: Average recommendation helpfulness score for different participants

Figure 4.10: Distribution of categories of recommended web resources

notation, and sqlite3 in Python, and the Wikipedia page for “Don’t Repeat Yourself”), which makes the average recommendation helpfulness score for this seed hyperlink an outlier.

4.6.3 Perceived Diversity

Figure 4.10 presents the percentage of different categories of the 1,800 recommended web resources for the 180 seed hyperlinks. Among the 1,800 recommended web resources, official documentation accounts are about 50.9%, and other types of docu-

mentation (*i.e.*, encyclopedia (13.6%), unofficial documentation (13.2%), and Q&A site (10.4%)) account for about 37.2%. Code repository accounts for a small portion (4.5%). Others, including 44 dead links and some downloading links, account for 7.39%. Participants all rate web resources in others category as unhelpful.

Among the 180 sampled seed hyperlinks, LinkLive recommends 1 category of web resources for only 12.7% (23/180) seed hyperlinks, and recommends 2 or more categories of web resources for 87.3% (157/180) seed hyperlinks. For 7 seed hyperlinks, the recommend web resources cover all the five categories. For example, one of these 7 seed hyperlinks <http://scikit-learn.org/stable/index.html> links to Python *scikit-learn* machine learning library. LinkLive recommends 5 categories of web resources, including the Wikipedia page for regression analysis, the *countmotifs.py* project on Github, Stack Overflow posts about machine learning, *scikit-learn* official documentation about feature extraction, and a professor home page for a list of hand-written, face, text and speech datasets.

The LinkLive tool helps discover helpful and diverse web resources that may assist developers in understanding Stack Overflow discussions and/or content of seed hyperlinks.

4.6.4 Threats to Validity

Participants indicate that it is straightforward to select category for a recommended web resource. However, participants (*e.g.*, Participant1 versus Participant6) exhibit different rating behaviors for helpfulness of recommended web resources, because the evaluation of the helpfulness is based on subjective assessment, and is affected by priori knowledge (or absence of knowledge) of the participants. A general feedback from the participants is that it is sometimes difficult to determine the helpfulness of the recommended web resources because it depends on the information needs. For a developer looking for some basic knowledge, official documentation and encyclopedia-like information would be very helpful. But for a developer looking for a bug fix, a code example, a Stack Overflow post would be more helpful. That is, helpfulness is often context-sensitive. Due to these limitations, this study provides only initial evidence of the helpfulness of the LinkLive recommendation.

4.7 Discussion

In addition to hyperlinks, Stack Overflow discussions contain many other useful information, such as software-specific concepts like *machine learning*, *Observer pattern*, software tools and libraries like *Eclipse*, *Django*, and APIs like class names, method names. Several studies [Liu, 2009, Baeza-Yates et al., 2015, Subramanian et al., 2014], including recent work from our team [Ye et al., 2016c], propose software-specific named entity recognition techniques to recognize software-specific entities in Q&A discussions and other informal documentations. Once a rich set of software-specific entities has been recognized, our item-based collaborative filtering approach can be extended to make recommendation for a variety of entities that developer may be interested in.

An innovation of our approach is that we exploit crowdsourced knowledge in Stack Overflow (*i.e.*, hyperlink correlation patterns in this work) to support recommendation tasks beyond Q&A. This is different from existing recommendation systems [Parnin et al., 2012, Ponzanelli et al., 2013b, Anderson et al., 2012, Treude et al., 2011, Li et al., 2015] which mainly focus on facilitating access to online programming resources or social content in software development. The crowdsourced knowledge underlying our LinkLive recommendation could be enhanced to deliver entity-centric search services for software developers, similar to SimilarTech⁸, AlternativeTo⁹, or SimilarWeb¹⁰. Entity-centric search systems, such as serendipitous search system [Marchionini, 2006, Sakai and Nogami, 2009], direct answers [Seebach, 2012, Bernstein et al., 2012], entity-centric recommendation [Lin et al., 2012, Blanco et al., 2013], have been actively researched in information retrieval community. Our work is an attempt along this line of research for software engineering data.

The LinkLive recommendation relies on frequent hyperlink co-occurrences in Q&A discussions. We are now investigating neural-network-based deep learning techniques (such as [Tang et al., 2014, Zhou et al., 2015]) to mine semantically related web resources from the discussion context that they are referenced. Neural-network-based

⁸<https://www.similartech.com/>

⁹<http://alternativeto.net/>

¹⁰<http://www.similarweb.com/>

techniques have been successfully applied in many natural language processing applications to learn semantic representation of words based on the assumption that word with similar meaning tend to present in similar contexts. We hypothesize that semantically related web resources could be referenced in similar discussion contexts, even though they may not be frequently referenced together in the same discussion threads. Co-occurrence based recommendation and neural-network based recommendation could be complementary. Furthermore, embedding hyperlinks with the discussion context could enable context-sensitive recommendation of relevant web resources.

4.8 Summary

In this chapter, we present an item-based collaborative filtering approach for recommending correlated web resources like the ones that developers are interested in. Our approach exploits the fact that correlated web resources have been frequently referenced in discussions on Stack Overflow. Taken in aggregation, hyperlink correlation patterns can be discovered from Stack Overflow discussions for recommendation of community-recognized, correlated web resources. We implement a proof-of-concept tool, named LinkLive, and evaluate its recommendation quality in two studies. Our evaluation shows that LinkLive recommends helpful and diverse correlated web resources with satisfactory accuracy. In the future, we will investigate deep learning techniques to discover semantically correlated web resources, and extend our approach to recommendation of a variety of software-specific entities.

Learning to Answer Programming Questions with Software Documentation via Web Q&A Context Embedding

For a successful technology, reality must take precedence over public relations, for Nature cannot be fooled.

— Richard Feynman

Programming languages and software packages are often well supported through formal documentation. Official software documentation provides a comprehensive overview of software usages, but not on specific programming tasks or use cases. Often there is a mismatch between the documentation and the question stating specific programming tasks because of different wordings. We observe from Stack Overflow that the best answers to programmers' questions often contain links to formal documentation. In this Chapter, we propose a novel *deep-learning-to-answer* framework, named QDLinker, for answering programming questions with software documentation. QDLinker learns from the large volume of discussions in community-based question answering site to bridge the semantic gap between programmers' questions and software documentation. Specifically, QDLinker learns question-documentation semantic representation from these question answering discussions with a four-layer neural network, and incorporates semantic and content features into a learning-to-rank schema. Our approach does not require manual feature engineering or external resources to infer the degree of relevance between questions and documentation. Through extensive experiments, results show that QDLinker effectively answers programming questions with direct links to software documentation. QDLinker significantly outperforms the baselines based on the traditional retrieval models and Web search services dedicated for software documentation retrieval. The user study shows that QDLinker effectively bridges the semantic gap between the intent of programming questions and the content of software documentation.

5.1 Background and Motivation

For most programming languages and software packages, there exist comprehensive language specifications, Application Programming Interface (API) documentation, and tutorials. Such official documentation¹ provides information about functionality, structure, and parameters, but not on specific issues or specific usage scenarios [Nadi et al., 2016, Robillard and Deline, 2011]. On the other hand, programmers often face very specific issues which are not explicitly stated in software documentation. For many such issues, the software documentation does serve as good reference for why the issues happen and how to address them. However, searching for software documentation using a question as a keyword query is challenging, because software documentation and questions are in different wording; one is for generic reference and the others are from specific usage scenarios.

With the emergence of Web 2.0 in modern software development, the behavior of developers is changed in relation to how they search for crowd-generated knowledge online to fulfill their needs [Ko et al., 2007, Ko et al., 2006]. The mismatch between the needs of documentation consumers and the knowledge provided, leads to the overwhelming discussions accumulated at various Community-based Question Answering (CQA) websites such as Quora² and Stack Overflow³. In these discussions, users often refer to software documentation when answering programming questions. From Stack Overflow, we collected 45,288 best answers each contain at least one link to Java official documentation. Figure 5.1 plots the distribution of the number of links to Java documentation per best answer. It shows that 72.6% of best answers have exactly one link to Java documentation and fewer than 10% have more than three links. This distribution suggests that for many Java programming questions, there exist a Java official document as good reference to address the question. The large volume of discussions also create the ‘*semantic link*’ between programmers’ questions and documentation, through the community of programmers, illustrated in Figure 5.2.

¹The term ‘software documentation’ refers to the collection of documents consisting of language specification, API documentation, and official tutorial.

²<https://www.quora.com/>

³<http://stackoverflow.com/>

Figure 5.1: Distribution of links to Java documentation per best answer, among 45, 288 best answers from Stack Overflow

Posting questions and waiting for answers from other programmers may take much time. The immediate question is: *can we answer a programmer's question by providing a link to the most relevant software documentation?* In this research, we aim to build an answering system where the questions are from programmers in natural language and the answers are the links to official documentation, illustrated in Figure 5.2. This system will provide convenience not only for documentation consumers but also companies that provide technical support.

However, understanding programming questions to build an effective answering system is not a trivial matter for machines. First of all, mapping question-answer pairs into a discriminative feature space is a critical step. A widely adopted approach is to encode question-answer pairs using various features, *e.g.*, lexical, linguistic, and syntactic features [Zou et al., 2015, Petrosyan et al., 2015, Toba et al., 2014, Yen et al., 2013, Palomera and Figueroa, 2017]. The hand-crafted features may heavily depend on external resources and are not easily generalizable. Besides, many existing knowledge bases are about lexical knowledge or about open domain facts. A typical example is WordNet [Miller, 1995], a lexical knowledge base for general English language, which may be not suitable to build answer systems for technical questions

Figure 5.2: Overview of QDLinker. It directly links programmer’s question to formal documentation through embedding the semantic context in CQA. Before QDLinker, the semantic links between questions and software documentation are established through the CQA community.

about programming. As shown by the analysis above, taking the advantage of neural networks to learn semantic representation of question-documentation pair seems to be more appropriate for our task. Note that, our task cannot be addressed by search engines for source code [Gu et al., 2016, Niu et al., 2016]. Code search system cannot well answer queries in natural language, especially when the queries do not contain any code snippets or API-like terms.

In this chapter, we propose a novel *deep learning to answer* framework named QDLinker, to answer programming questions with software documentation through social context embedding. *Social context* of a link to software documentation refers to the surrounding words of the link, when community users use it to answer questions in CQA. QDLinker embeds social contexts in a latent space, and uses a four-layer Deep Neural Network (DNN) to learn semantic representation of question-documentation pairs. The learned semantic representations and simple content features are then passed to a learning-to-rank schema to train a ranker. Compared with previous work on software text retrieval [Zou et al., 2015], our approach does not require feature engineering or hand-coded resources beyond the pre-trained word vectors. The architecture we proposed can benefit not only learning a ranker in training phase, but

also automatic feature extraction for newcoming query-documentation pairs in online phase. Moreover, our approach takes into account documentation content and social context simultaneously, for its effectiveness in bridging the semantic gap between programming questions and software documentation.

We conducted extensive experiments on Stack Overflow dataset to evaluate the effectiveness of QDLinker. Empirical results show that QDLinker outperforms three baseline methods which are based on traditional retrieval models. Through a user study with 25 natural language queries collected from test dataset, we show that QDLinker significantly outperforms a commercial search engine. In short, our empirical results show that QDLinker can effectively bridge the semantic gap between questions and software documentation. In this chapter, we make the following contributions:

- We propose QDLinker, a novel framework for answering programming questions with software documentation through social context embedding. It leverages the content in official sites and social context in CQA to learn semantic representation of question-documentation pairs and answers programming questions in natural language.
- We conduct a large-scale automatic evaluation, to evaluate the performance of QDLinker against three baseline methods. The empirical evaluation reveals that our approach can effectively answer Java technical questions against traditional retrieval models.
- We conduct a user study to compare the software documentation retrieval performance by QDLinker with Google search. QDLinker significantly outperforms Google search in the retrieval task.

5.2 Related Work

Question Retrieval. Question retrieval has attracted much attention in recent years [Cao et al., 2010, Duan et al., 2008, Jeon et al., 2005, Figueroa and Neumann, 2016]. Different retrieval models have been employed in the task, including the Okapi model [Jeon et al., 2005], the translation model [Zhou et al., 2011], the language model [Duan et al., 2008], and the vector space model [Jeon et al., 2005, Ji et al., 2012].

In addition, question category information has also been exploited for question retrieval [Cao et al., 2010]. Xue et al. [Xue et al., 2008] proposed a translation-based language model that combines the translation model and the language model for question retrieval. Yen et al. [Yen et al., 2013] developed a question classifier, which is trained to categorize the answer type of the given question and instructs the context-ranking model to re-rank the passages retrieved from the initial retrievers.

For the word mismatch problem among similar questions, existing solutions can be broadly grouped into three approaches to bridge the lexical gap. One approach is to use manual rules or templates. For example, Berger et al. [Berger et al., 2000] proposed a statistical lexicon correlation method to bridge the lexical chasm. The second approach is to use external knowledge databases such as Wikipedia and WordNet. The method by Zhou et al. [Zhou et al., 2013] using semantic relations extracted from Wikipedia for question retrieval is an example. Burke et al. [Burke et al., 1997] proposed a model to rank frequently asked questions using combined similarities. The similarities are computed by conventional vector space models with semantic similarities based on WordNet. The third approach is to use deep representation. Zhou et al. [Zhou et al., 2015, Zhou et al., 2016] proposed a neural network architecture to learn the semantic representation of question-answer pairs. Nassif et al. [Nassif et al., 2016] presented a neural-based model with stacked bidirectional Long Short-Term Memory (LSTM) and Multi-Layer Perceptron (MLP) for similar question retrieval. Different from the earlier studies, we aim to directly link questions with software documentation, rather than retrieving similar questions.

Answer Selection. Given a thread containing a question and a list of answers, many studies aim to automatically rank the answers according to their relevance to the question [Sun et al., 2005, Sakai et al., 2011, Surdeanu et al., 2008]. Sun et al. [Sun et al., 2005] used dependency relations between the matched question terms and the answer target as additional evidence to rank passages. Sakai et al. [Sakai et al., 2011] proposed an approach to build answer selection systems involving multiple answer assessors and graded-relevance information retrieval metrics. Yao et al. [Yao et al., 2015] proposed a family of algorithms to jointly detecting the high-quality questions and help users to identify a useful answer that would gain much positive feedback

from site users. Hou et al. [Hou et al., 2015] and Nicosia et al. [Nicosia et al., 2015] proposed automatic answer selection algorithms based on the position of the answer in the thread and the context of an answer in a thread, respectively. Instead of selecting answers, in our task, we attempt to distill the software documentation in answers.

CQA Semantic Representation. Most relevant to our work is the study on semantic representation of CQA. In recent years, deep neural networks have been used to learn higher-level semantic representation of question answer pairs [Severyn and Moschitti, 2015, Nassif et al., 2016, Severyn and Moschitti, 2016, Er et al., 2016, Kokkinos and Margaritis, 2015]. Tan et al. [Tan et al., 2016] developed hybrid models to match passage answers to questions accommodating their complex semantic relations. Severyn et al. [Severyn and Moschitti, 2015] proposed a convolutional neural network architecture, which maps queries and documents to their distributional vectors, for reranking pairs of short texts. Yan et al. [Yan et al., 2016] proposed a deep neural network to learn how a query and its context are related to candidate reply. Nassif et al. [Nassif et al., 2016] presented a neural-based model with stacked bidirectional LSTMs and MLP to learn semantic relatedness between questions and answers. Singh et al. [Singh and Simperl, 2016] proposed a system using semantic keyword search in combination with traditional text search techniques to find similar questions with answers for unanswered questions.

In our proposed QDLinker, we learn the semantic representation of programming questions and the social context of software documentation. In our case, the social context, *i.e.*, how the APIs are used in different scenarios, cannot be obtained through the content of software documentation itself.

5.3 Deep Learning to Answer

In this section, we first give an overview of proposed framework QDLinker, and then detail the core modules in QDLinker in Sections 5.3.2 - 5.3.4. The input to the framework, *i.e.*, the word embedding, is presented in Section 5.3.1.

Shown in Figure 5.3, the QDLinker framework consists of three core modules: *candidate document generation*, *a four-layer neural network*, and *learning a ranker*. Given a

Figure 5.3: The architecture of our proposed approach

programming question in natural language, *candidate document generation* returns a small set of software documents which are considered relevant to the question. The DNN module learns the semantic representation of query-documentation pairs in a latent space, and generates features for the ranker module. The features by DNN are fully automatic without human intervention. Nevertheless, the network does allow handcrafted features to be inserted in the join layer, illustrated in Figure 5.3. The learned features are then fed to a learning-to-rank schema to train a ranker, to pick up the more relevant software documents among the candidates.

Note that our learning-to-rank takes lists (not pairs) as instances in learning and loss function of ranker is defined on input list. There are some reasons that we did not solve the end-to-end problem as minimizing directly a ranking cost function. (1) The multiple instances in input list will result in multiple channels in DNN architecture, which dramatically increases network complexity but there is no performance improvement based on our experiments. (2) If we treat our problem as an end-to-end problem in ranking cost function in offline phase, we cannot automatically extract features for newcomers candidates in online phase. Thus, the DNN only serves as extracting the final abstract representation of the query-documentation pair. This setup can benefit not only learning a ranker in offline phase, but also automatic feature extraction for new query-documentation pairs in online phase.

Figure 5.4: Training word embedding example with the skip-gram model

5.3.1 Social Context Embedding

As shown in Figure 5.3, QDLinker is built on pre-trained word vectors, or word embeddings. Traditionally, language models represent each word as a feature vector using one-hot representation, where a vector element is 1 if the word is observed and 0 otherwise [Grbovic et al., 2015a]. Recently, neural language models have been proposed to generate low-dimensional, distributed embedding of words [Collobert et al., 2011, Turian et al., 2010]. These models take advantage of word order in text documents and capture both syntactic and semantic relationships between words. Mikolov’s continuous bag-of-words and skip-gram language models [Mikolov et al., 2013a, Mikolov et al., 2013b] are among the most widely used models.

Stack Overflow is a destination with rich source of information about API usages and bug descriptions. Thus, in our implementation, we use the crowd-generated content in Stack Overflow to learn embeddings of the words and the links to software documentation.

As shown in Figure 5.4, a community user of Stack Overflow created a link to API documentation `java.util.Collections.sort()` in an answer. Figure 5.4 illustrates the training procedure with the skip-gram model when it reaches the link. This sentence creates a context for the link `java.util.Collections.sort()` through the surrounding words. We build two vocabularies: one for English words, and the other for the links to software documentation. In simple words, each link to software documentation is treated as an ordinary term in the word sequence, and a word vector is

learned for each link that is mentioned in Stack Overflow. Note that, we learn word embedding for each link as a term, and the words in the anchor text of the link are not used in our training.

We use V_{w_t} to denote the input of vector of the only word w_t on the input layer. N is the hidden layer size. V is the vocabulary size. C is the number of words in the context. The output of hidden layer can be written as

$$\mathbf{h} = \mathbf{W}^T \mathbf{x} = \mathbf{V}_{w_t}^T \quad (5.1)$$

where $\mathbf{W}$ is a $V \times N$ input $\rightarrow$ hidden weight matrix. $\mathbf{V}_{w_t}$ is the vector representation of the input word w_t .

On the output layer, each output is computed using the hidden $\rightarrow$ output matrix:

$$p(w_{c,j} = w_{O,c} | w_t) = \frac{\exp(u_{c,j})}{\sum_{j'=1}^V \exp(u_{j'})} \quad (5.2)$$

where w_t is the input word. $w_{c,j}$ is the j -th word on the c -th panel of the output layer. $w_{O,c}$ is the actual c -th word in the output context word. $u_{c,j}$ is the net input of the j -th unit on the c -th panel of the output layer,

$$u_{c,j} = \mathbf{V}_{w_j}^T \cdot \mathbf{h}, \text{ for } c = 1, 2, \dots, C \quad (5.3)$$

where $\mathbf{V}_{w_j}^T$ is the output vector of the j -th word in the vocabulary, w_j and $\mathbf{V}_{w_j}^T$ is taken from a column of the hidden $\rightarrow$ output weight matrix, $\mathbf{W}'$.

When training the skip-gram model to predict C context words, the loss function is written as

$$\begin{aligned} E &= -\log p(w_{O,1}, w_{O,2}, \dots, w_{O,C} | w_t) \\ &= -\log \prod_{c=1}^C \frac{\exp(u_{c,j_c^*})}{\sum_{j'=1}^V \exp(u_{j'})} \end{aligned} \quad (5.4)$$

where j_c^* is the index of the actual c -th output context word in the vocabulary.

Figure 5.5 illustrates a 2-D projection of vectors of natural language terms and API documentation using principal component analysis (PCA). In the embedding space, the vectors of terms and API documentation with same intent have the shortest distance. For example, the term “arraylist” is close to API documentation `java.util.ArrayList`. API documentation `java.awt.Window.pack()` is close to `java.swing.JFrame`.

Figure 5.5: A 2D projection of embedding natural language terms and API documentation using PCA (API documentation in bold font and natural language terms in non-bold font)

5.3.2 Candidate Document Generation

Given a programming question as a query, candidate generation selects a subset of software documents that are relevant to the question. Considering there are three methods to select candidates, we thus select top-10 candidates for each method.

Document Content. The content of software document reflects its relevance to a given query. In our implementation, we build a search engine for software documentation using Apache Lucene. Specifically, stopwords removal and stemming are performed as preprocessing, and for each query, the search engine returns top 10 most relevant results based on the BM25 scoring function.

Local Context. Stack Overflow is a popular CQA site where developers ask questions and share knowledge about software development and maintenance. The discussions in Stack Overflow provide enriching context to mine usage scenarios of software documentation. When a software document appears in a discussion thread, its surround texts reflect its relevance to the query question.

Definition 5.8 (Local Context) *If a software document is mentioned in a best answer, the texts of the question (title and body) and the best answer are regarded as the local context of the software document.*

Local context is defined based on the consideration that the quality of the best answer is better than other answers in the discussion thread, to avoid including too much noise. The body of the best answer is the immediate context where a software document is mentioned. On the other hand, question title and body often describe the programming issues and reflect the relevance between the problem and the software document mentioned in its best answer.

Note that, each mention of a software document has its own local context. If a software document is mentioned multiple times, multiple local contexts are extracted. We collect all local contexts of the mentioned software documents in our corpus. Given a query, we use Lucene to retrieve the most relevant local contexts, then pick the top 10 unique software documents as candidates.

Global Context. As aforementioned, a software document may be mentioned in multiple best answers and has multiple pieces of local context. For example, the API `java.util.ArrayList` was mentioned 906 times in our dataset.

Definition 5.9 (Global Context) *The global context of a software document is the collection of all its local contexts.*

We build up a corpus through collecting global context for all software documents. Then we obtain the vector of a software document by social context embedding described in Section 5.3.1. Following [Van Nguyen et al., 2016], we use bag-of-words model to average out the vectors of the individual words in a query. Given a query, we retrieve top 10 software documents based on the cosine similarity between the average vector and software documentation vector.

5.3.3 Four-layer Deep Neural Network

Deep neural networks with multiple layers have been demonstrated its effectiveness in capturing semantical and higher-level discriminative information from the input

data [LeCun et al., 2015]. As shown in Figure 5.3, our DNN has four layers: *convolutional layer*, *join layer*, *hidden layer*, and *output layer*.

(1) Convolutional Layer

For a natural language query with many words in a sentence, earlier studies [Yu et al., 2014, Kim, 2014] have shown that the simple bag-of-words model is unable to capture complex semantics of a sentence. Convolutional neural network can capture long-range dependencies and learn to correspond to the internal syntactic structure of sentences. Thus, we use one convolutional layer as the first layer in our approach.

Convolution Operation. Let $\mathbf{X}_d$ and $\mathbf{X}_q$ be the software documentation vector and query vector, respectively. Suppose that there are s words in the query and let $\mathbf{X}_q^i \in \mathbb{R}^k$ be the i -th k -dimensional word vector corresponding to the i -th word in the query. More formally, the convolution operation $*$ between two vectors $\mathbf{X}_q \in \mathbb{R}^{sk}$ and $\mathbf{f}_q \in \mathbb{R}^{mk}$ (called a filter of size m) results in a vector $\mathbf{c}_q \in \mathbb{R}^{s-m+1}$ where each component is as follows:

$$c_q^j = (\mathbf{X}_q * \mathbf{f}_q)_j = \mathbf{f}_q^T \cdot \mathbf{X}_q^{[j:j+m-1]} + b_{qc} \quad (5.5)$$

where $j = 1, \dots, s - m + 1$ and $\mathbf{X}_q^{[j:j+m-1]}$ represents the concatenation word vectors $\mathbf{X}_q^j, \mathbf{X}_q^{j+1}, \dots, \mathbf{X}_q^{j+m-1}$, $b_{qc} \in \mathbb{R}$ is a bias term. Thus, this filter $\mathbf{f}_q$ is applied to each possible window of words in the query to produce a feature map:

$$\mathbf{c}_q = [c_q^1, c_q^2, \dots, c_q^{s-m+1}] \quad (5.6)$$

where $\mathbf{c}_q \in \mathbb{R}^{s-m+1}$. Similarly, we can utilize filter $\mathbf{f}_d$ to produce documentation feature map $\mathbf{c}_d$.

So far we have described the convolution layer with a single filter. Our model applies a set of filters that work in parallel, and generates multiple feature maps. Let n be the number of filters. Given filters $\mathbf{F}_q^{n \times mk}$ and $\mathbf{F}_d^{n \times mk}$, the convolution operations produce two feature maps $\mathbf{C}_q \in \mathbb{R}^{n \times (s-m+1)}$ and $\mathbf{C}_d \in \mathbb{R}^{n \times (s-m+1)}$, respectively.

Activation Function. To allow the neural network to learn non-linear decision boundaries, each convolutional layer is followed by a non-linear activation function applied element-wise to the output of the convolution operations. Sigmoid, hyperbolic tangent $\tanh$, and a rectified linear ($ReLU$) are among the most common choices for

activation functions. In particular, it is reported that rectified linear unit has significant benefits over sigmoid and $\tanh$ functions [Nair and Hinton, 2010]. Thus, in our implementation, we use $ReLU$ as the activation function. The output of activation layer can be written as

$$\mathbf{A}_q = ReLU(\mathbf{C}_q) = \max(0, \mathbf{C}_q) \quad (5.7)$$

$$\mathbf{A}_d = ReLU(\mathbf{C}_d) = \max(0, \mathbf{C}_d) \quad (5.8)$$

where $\mathbf{A}_q \in \mathbb{R}^{n \times (s-m+1)}$ and $\mathbf{A}_d \in \mathbb{R}^{n \times (s-m+1)}$.

Pooling. The output from activation function is then passed to the pooling layer, whose goal is to aggregate the information and reduce the representation. As mentioned above, there are n filters. The pooling operation is applied on every filter. Taking the pooling of $\mathbf{A}_q \in \mathbb{R}^{n \times (s-m+1)}$ as an example, the output of pooling $\mathbf{P}_q \in \mathbb{R}^n$ can be written as

$$\mathbf{P}_q = \begin{bmatrix} \text{pool}(\mathbf{A}_q^1) \\ \vdots \\ \text{pool}(\mathbf{A}_q^n) \end{bmatrix} \quad (5.9)$$

The pooling operation maps the feature map to a single value, formally: $\text{pool}(\mathbf{A}_q^i) : \mathbb{R}^{1 \times (s-m+1)} \rightarrow \mathbf{P}_q^i : \mathbb{R}$. There are a few common choices for the $\text{pool}()$ operations: *average*, *max* and *L2-norm*. Average pooling was often used historically but has recently fallen out of favor compared to the max pooling operation, which has been shown to work better in practice. In our approach, we use 1-*max* pooling strategy, which extracts a scalar with the maximum value for each feature map.

(2) Join Layer

Inspired by [Yu et al., 2014, Severyn and Moschitti, 2015], we also add simple content features $\mathbf{f}_{cn}$ to our model. $\mathbf{f}_{cn}$ contains two word overlap features: word overlap count, and word overlap count weighted by IDF (inverse document frequency). Note that both features do not require any linguistic annotation or pre-processing. The output of join layer $\mathbf{X}_{join} \in \mathbb{R}^{2n+2}$ can be expressed as follows:

$$\mathbf{X}_{join} = [\mathbf{P}_d; \mathbf{P}_q; \mathbf{f}_{cn}] \quad (5.10)$$

(3) Hidden Layer

DNNs could use the intermediate layers to build up multiple layers of abstraction. These multiple layers of abstraction seem likely to give deep networks a compelling advantage in learning to solve complex pattern recognition problems [Schmidhuber, 2015]. In our architecture, the hidden layer is a fully-connected layer with parameters $\mathbf{W}_h$ and $\mathbf{b}$. The output of hidden layer can be represented as

$$\mathbf{X}_{hidden} = ReLU(\mathbf{W}_h \cdot \mathbf{X}_{join} + \mathbf{b}_h) \quad (5.11)$$

(4) Output Layer

The output of hidden layer $\mathbf{X}_{hidden}$ is passed to a fully connected softmax layer. It computes the probability distribution over the class labels:

$$p(y = j | \mathbf{X}_{hidden}; \mathbf{W}_s, \mathbf{b}_s) = softmax_j(\mathbf{W}_s \cdot \mathbf{X}_{hidden} + \mathbf{b}_s) \quad (5.12)$$

where $\mathbf{W}_s$ and $\mathbf{b}_s$ are the weight vector and the bias of softmax classifier, respectively. Our model is trained to minimize the cross-entropy cost function:

$$\mathcal{L} = -\log \prod_{i=1}^N p(y_i | \mathbf{X}_q^i, \mathbf{X}_d^i) + \lambda \|\theta\|_2^2 \quad (5.13)$$

where θ contains all parameters and we use *L2-norm* regularization.

$$\theta = \{\mathbf{F}_q, \mathbf{b}_{qc}, \mathbf{F}_d, \mathbf{b}_{dc}, \mathbf{W}_h, \mathbf{b}_h, \mathbf{W}_s, \mathbf{b}_s\} \quad (5.14)$$

In our problem setting, for a given question-documentation pair as an input instance, softmax layer outputs probabilities for two classification labels: positive and negative. Figure 5.6 shows an example question-documentation pair extracted from discussions in Stack Overflow. Together with the question, each link to documentation mentioned in the question's best answer forms a positive question-documentation pair instance.

For training the DNN, we use links from the best answers of the training questions to form positive instances, and use randomly select links to form negative instances. We use backpropagation algorithm to compute the gradients and use Adam update rule [Kingma and Ba, 2014] to update the parameters of the network. To mitigate the overfitting issue we augment the cost function with *L2-norm* regularization for the parameters of the network.

Figure 5.6: An example of extracting query-documentation pair from discussion thread in Stack Overflow

5.3.4 Learning a Ranker

In particular, $\mathbf{X}_{hidden}$ can be thought of as a final abstract representation of a query-documentation pair, obtained by a series of transformations from the input layer through a series of layers. In our approach, we consider $\mathbf{X}_{hidden}$ as features to feed to a learning-to-rank schema. The learning-to-rank schema can leverage multiple features for ranking and automatically learn the optimal way of combining these features.

Our goal is to build a ranking model which facilitates each query q and its candidate list $D = \{d_1, d_2, \dots, d_n\}$ to generate the optimal ranking. More formally, the task is to learn a scoring function $F(q, d)$:

$$F(q, d) = \sum_{k=1}^K \omega_k \cdot \phi_k(q, d) \quad (5.15)$$

where each feature $\phi_i(q, d) \in \mathbf{X}_{hidden}$ measures a specific relationship between the query and a candidate software document. ω_i is the weight of the i -th feature (among

Figure 5.7: The flow diagram of the proposed QDLinker. The middle part is to learn a model for extracting abstract representation for question-documentation pair. The left part aims to learn a ranker. The right part is to answer a new question with the learned models.

the total K features), and is learned during the training process. In our task, the optimization procedure of learning-to-rank tries to find the scoring function that ranks the most relevant software document to the query at the top among all candidates. We train the ranking model using LambdaMART [Wu et al., 2010], a boosted tree version of LambdaRank [Quoc and Le, 2007], that won the Yahoo! Learning to Rank Challenge.

5.3.5 Summary and Complexity Analysis

To summarize, we present a flow diagram of QDLinker in Figure 5.7 and the pseudocode in Algorithm 1 and Algorithm 2. Observe from Figure 5.7, our framework contains both offline phase and online phase. The offline phase first learns abstract representation of question-document pair and then learn a ranker based on the positive and negative instances. Here an instance is a question-document pair. The offline phase is listed in Algorithm 1. Given a new question, the online phase first generates the k candidate software documents based on Section 5.3.2. Then these documents are

Algorithm 1: Pseudocode for offline phase of the proposed QDLinker

Input: N training instances of question-document pairs
Output: Feature vector function $V_\theta(q, d)$ and ranking function F
// Phase 1: Learning abstract representation

```

1 Initialize all parameters  $\theta$ ;
2 foreach  $epoch$  in  $epoch_{max}$  do                                // iterate through epoches
3   Sample a minin-batch from  $N$  pairs;
4   Clear gradients  $d\theta \leftarrow 0$ ;
5   Computing  $\mathcal{L}$  based on Equation (5.13);
6   Update  $\theta \leftarrow \theta - \frac{\partial \mathcal{L}}{\partial \theta} \cdot lr$ ;
7   return  $V_\theta(q, d)$ ;                                // return feature vector function for q-d pair

// Phase 2: Learning a ranker
8 Set number of trees  $M$ , number of leaves per tree  $L$ ;
9 foreach  $m$  in  $M$  do                                // iterate trees
10  foreach  $n$  in  $N$  do                                // iterate through training pairs
11    Calculating feature vector for current  $n$  via  $V_\theta(q, d)$  in Phase 1;
12    Calculating the  $\lambda$ -gradients for each q-d pair;                // more detials[Wu et al., 2010]
13    Calculating the second-order derivative using the  $\lambda$ -gradients;
14    Building a regreesion tree with  $L$  terminal nodes;
15    Update  $F$  function;
16  return  $F$ ;                                // Final ranking function  $F$ 
```

Algorithm 2: Pseudocode for online phase of the proposed QDLinker

Input: A new question q_{new}
Output: top- k software documents

```

1 Retrieving candidate documents for  $q_{new}$  based on Section 3.2;
2 Extracting features for these question-document pairs using  $V_\theta(q, d)$  in Phase 1 of Algorithm 1;
3 Ranking these candidate documents using  $F$  in Phase 2 of Algorithm 1;
4 return top- $k$  software documents;                // Answers to the issued question
```

ranked by the learned models of offline phase. Accordingly, the online phase is listed in Algorithm 2.

We first detail the parameter size of our framework, then deduce the time and space complexity. Equations (5.14) and (5.15) show all parameters that QDLinker should learn. Now we consider one query-document pair as shown in Figure 5.3. For convolutional layer, QDLinker has $2n \times mk + 2n$ parameters, where n is the number of filters and m is the window size of each filter, k is the dimension of word embedding, and number 2 indicates query channel and document channel. For join layer, there is no parameter. For hidden layer, there are $(2n + 2) \times h + h$ parameters, where h is the number of neruons in hidden layer. For output layer, there are $h \times 2 + 2$ parameters. For ranker layer, there are h weights because QDLinker uses the $\mathbf{X}_{hidden}$ as the final abstract representation.

Given a new question, we assume that QDLinker generates κ candidate documents. Now considering a question-document pair, the total time and space complexity of

convolutional layer are both $O(\alpha \cdot m^2 \cdot \beta)$ [Cheng et al., 2015, He and Sun, 2015], where α and β are the number of input nodes and the number of output nodes respectively, m is the window size of each filter. Hidden layer and output layer are fully connected layers. The time and space complexity of this linear projection are both $O(\alpha\beta)$. For the ranker, the time complexity is $O(\kappa^2)$ [Wu et al., 2010]. Thus, the time complexity of the QDLinker is $O(\kappa\alpha m^2\beta + \kappa^2)$, and the space complexity is $O(\alpha m^2\beta)$.

5.4 Empirical Evaluation

We now evaluate the effectiveness of QDLinker by measuring its accuracy on linking questions to software documentation in Stack Overflow. Our evaluation assumes that the software documentation mentioned in a question’s best answer is most relevant to the question.

5.4.1 Experimental Setting

Data Collection. In our evaluation, we focus on Java software documentation which consists of Java Standard Edition API documentation, Java tutorials, and language specifications. Usually, a programming question is expressed in natural language thus it is similar to the discussions in Stack Overflow. We therefore use data collected from Stack Overflow in our experiments.

We extract discussion threads from the datadump archive⁴ that satisfy the following criteria: (i) The score of question is greater than 0. This condition guarantees that at least one developer has voted the question to be a ‘useful question’. (ii) The question has an answer which is accepted as the best answer, and the score of the best answer is greater than 0, and (iii) The best answer must contain at least one link to the above listed Java documentation. Based on the above criteria, we collect 30,272 discussion threads from the data dump released on August 2015. We randomly select 24,217 discussion threads (account for 80%) as training data and the remaining 6,055 threads (account for 20%) as test data.

⁴<https://archive.org/details/stackexchange>

Table 5.1: Summary of the number of threads used in each task

Data	#Discussion threads
Word embeddings	24,217
Train	10,649
Development	1,000
Test	1,693

Model Training. We learn word embeddings from the training data, *i.e.*, the 24,217 discussion threads. For each discussion thread, we extract text from question title, question body, and all answers whose scores are greater than 0. We use the skip-gram model implemented in word2vec⁵. The context window size is set to 10 and the minimal word frequency is 5. Recall that each link to a software document is also treated as a word (or term, see Figure 5.4). Based on this condition, we have 1,520 distinct links to Java documentation in the training data.

Next is to train the DNN and the ranker. Note that some discussion threads cannot be used to extract query-documentation pairs for training DNN because the links to software documentation in best answers are filtered out when the minimal word frequency is set by 5. Finally, we have 10,649 discussion threads for training DNN and the ranker, 1,000 discussion threads used for development set, and 1,693 discussion threads used for test. Table 5.1 summarizes the dataset in our experiments.

We empirically set the hyperparameters based on the development set. The number of filters in convolutional layer is 64, and the size of filter is set to 2. The size of hidden layer is set to 64. The dimensionality of pre-trained word vectors is 200. $L2$ -norm term is set to $1e - 5$ and the learning rate is $1e - 3$.

Performance Measures. We use the following five performance metrics in our evaluation:

- Precision at k , $P@k = \frac{|D_k \cap D_g|}{k}$, is the fraction of relevant documentation links to the query question among the top k ranked results. D_k denotes the set of top- k ranked links to software documentation and D_g is the set of ground-truth links (*i.e.*, links to software documentation in the question’s best answer).

⁵<https://code.google.com/archive/p/word2vec/>

- Recall at k , $R@k = \frac{|D_k \cap D_g|}{|D_g|}$ is the fraction of ground-truth links in the top- k results.
- Hit rate at k , denoted by $HR@k$, is 1 if $|D_k \cap D_g| > 0$ and 0 otherwise.
- Mean average precision, MAP , is the mean of average precision (AP) over a set of test queries.
- Mean reciprocal rank, $MRR(Q) = \frac{1}{|Q|} \sum_{j=1}^{|Q|} \frac{1}{rank_j}$, is the average reciprocal rank of the results over a set of test queries Q . In the equation, $rank_j$ denotes the rank position of the first relevant document for the j -th test query.

Baseline Methods. In order to validate the effectiveness of the proposed method, we evaluate the following three baseline methods in our experiments.

- OfficialCn: This is the baseline model which selects candidate software documentation by content (see Section 5.3.2). BM25 [Robertson et al., 1995] scoring function is used to rank the candidates.
- LocalCx: This baseline ranks candidates based on local context (see Section 5.3.2). This model indexes the local contexts in discussion threads and retrieves candidates using BM25 scoring.
- GlobalCx: This baseline ranks candidates based on global context (see Section 5.3.2). This model represents natural language words and software documentation as vectors in shared embedding space [Mikolov et al., 2013a].

5.4.2 Performance Comparison

Table 5.2 and Table 5.3 report the linking performance by different methods on all evaluation metrics. From the results, QDLinker significantly outperforms all baselines on all metrics. The improvement is statistically significant based on t -test with $p < 0.05$. Observe that $P@10$ is very small for all methods including QDLinker. In Stack Overflow, more than 70% of the best answers contain only one link to software documentation (reported in Section 5.1, Figure 5.1). As the result, $|D_{10} \cap D_g| = 1$ in

Table 5.2: Performance ($P@k$, $R@k$, MAP and MRR) for different methods. The best performance is highlighted in bold face. † indicates that the differences between the result of QDLinker and other models are statistically significant with $p < 0.05$ under t -test.

Method	P@1	P@2	P@5	P@10	R@1	R@2	R@5	R@10	MAP	MRR
OfficialCn	0.1347	0.1087	0.0749	0.0511	0.1147	0.1797	0.3053	0.4108	0.2234	0.2267
LocalCx	0.1630	0.1337	0.0932	0.0657	0.1401	0.2253	0.3876	0.5361	0.2896	0.2956
GlobalCx	0.1536	0.1234	0.0875	0.0628	0.1300	0.2002	0.3548	0.5005	0.2564	0.2614
QDLinker	0.1875†	0.1576†	0.1272†	0.0919†	0.1708†	0.2734†	0.5012†	0.6847†	0.3461†	0.3584†

Table 5.3: Performance($HR@k$) for different methods

Method	HR@1	HR@2	HR@5	HR@10
OfficialCn	0.1347	0.2069	0.3491	0.4663
LocalCx	0.1630	0.2631	0.4341	0.5914
GlobalCx	0.1536	0.2383	0.4109	0.5640
QDLinker	0.1875†	0.3057†	0.5828†	0.8128†

most cases. Thus, the ideal value of $P@10$ is slightly above 0.1, QDLinker achieves a very good result of 0.0919 in this sense. On $R@k$ measure, QDLinker significantly outperforms the other baselines for all k values ($k = 1, 2, 5, 10$). It is worth noting that QDLinker achieves the highest recall (0.6847) when $k = 10$. In terms of MAP measure, QDLinker outperforms the three baseline methods OfficialCn, LocalCx and GlobalCx by 34.98%, 19.51% and 54.93%, respectively. On MRR measure, QDLinker outperforms the three baselines by 37.11%, 21.24% and 58.09%, respectively. Similar observations hold on hit rate measure $HR@k$, reported in Table 5.3.

Observe from the results that the content-based method OfficialCn delivers the worst performance on all measures. This implies that content-based approach cannot bridge developers' intent and content of software documentation. This observation is consistent with our description in Section 5.1 that the software documentation is prepared to give comprehensive coverage without targeting on specific problems, while the programming questions are encountered in specific programming tasks. Therefore, it is essential to utilize the social context available at Stack Overflow to bridge the semantic gap between programmers' questions and software documentation.

Table 5.4: Results on additional content features

Content features	MAP	MRR
Without f_{cn} features	0.3054	0.3102
With f_{cn} features	0.3461($\uparrow$ 13.32%)	0.3584($\uparrow$ 15.53%)

Figure 5.8: Impact of dimensionality of word embedding

5.4.3 Impacts of Factors on Performance

Impact of content features. In our approach, f_{cn} consists of word overlap count and word overlap count weighted by IDF value. Table 5.4 shows the performance and improvement of considering additional content features f_{cn} . More specifically, considering f_{cn} improves MAP by 13.32%, and MRR by 15.53%.

There are two aspects resulting in the improvement. First, as described above, input of our approach are the pre-trained distributional word vectors. In fact, the word dictionary of our dataset may not cover all the words of English language. The overlap features can provide supplementary information in our approach. Additionally, one of the weaknesses of approaches relying on distributional word vectors is their inability to deal with numbers and proper nouns [Yu et al., 2014, Severyn and Moschitti, 2015]. But when developers issue natural language queries, most of the questions are of type what, when, who that are looking for answers containing numbers or proper nouns. Thus, the model with f_{cn} features outperforms the one without f_{cn} features on MAP and MRR .

Figure 5.9: Performance in MAP with different layer sizes

Dimensionality of word embedding. QDLinker takes in pre-trained word vectors in the input layer and feeds into the convolutional layer. We vary the dimensionality of word embedding and evaluate its impact on MAP and MRR . Figure 5.8 reports the performance QDLinker using word embedding in different dimensions (50, 100, 200, 300, 400, 500 and 600). The results indicate that the dimensionality of word embedding has very marginal impact on the performance.

The possible reason is that the distributional word vectors varying different dimensions contain enough latent information for building a ranker in our dataset. Thus, when training a ranker, our approach is stable for dimensions of word embedding.

Impact of layer sizes. As shown in Figure 5.3, our architecture needs to set the number of neurons (*i.e.*, layer size) in convolutional layer and hidden layer. Figure 5.9 shows the impact of setting different convolutional layer sizes and hidden layer sizes, on the test set. The comparison is based on 200 dimensions of word vectors.

We can observe that performance of ranker greatly depends on the combination of convolutional layer size and hidden layer size. In our dataset, we obtain the best performance when the convolutional layer size and hidden layer size are both 64.

When the combination layer sizes are small, the *MAP* is around 0.2 only, much lower than the best performance. Too few neurons in the convolutional and hidden layers will result in underfitting, as the neurons cannot capture enough signals to model complex data set. However, larger combination layer sizes does not lead to better ranker performance either, because of overfitting. In addition, a large number of neurons in the convolutional and hidden layers increases the training time.

In summary, dimensionality of word vectors has very marginal impact on the performance. However, the size of convolutional and hidden layers has significant impact on the performance of our model.

5.5 User Study

To the best of our knowledge, there is no existing work on answering programming questions in natural language. Commercial search engines, *e.g.*, Google and Bing, are tools for daily use in software development. It naturally motivates us to compare the returned results with these search engines. If we can improve the performance of search results on the search engines, it will provide convenience not only for developers but also for the companies that provide documentation support.

In the previous set of experiments, we consider the software documentation mentioned in the best answers as the ground truth to questions. This assumption may ignore the other retrieved software documentation which is relevant to the question but is not mentioned in the best answers. That is, although limiting documentation mention in best answer is a good criterion to control the quality of ground truth, the criterion may exclude the relevant results from our evaluation. In this section, we perform a user study to manually evaluate performance of QDLinker against Web search services.

5.5.1 Evaluation Setup

From the test dataset, we randomly select 25 discussion threads and query QDLinker using the questions. The 25 questions are listed in Table 5.5. For each question, we also use Google search engine to retrieve a list of software documentation. Because

Table 5.5: Human evaluation for Google search and QDLinker (*BA*: the number of ground truth links in best answer. *FR*: the rank of the first relevant documentation. *P@5* and *P@10*: precision of the first 5 and 10 results. “*A_*” indicates Java API documentatins. “*S_*” indicates Java language specification. “*T_*” indicates Java tutorial. The boldface documentation indicate the ground truth documentation in best answers. † indicates the differences between QDLinker and Google are significant with $p < 0.05$ under *t*-test).

Query	Google Search				QDLinker			Top 3 relevant documents by QDLinker
	BA	FR	P@5	P@10	FR	P@5	P@10	
1: java switch error when simplifying code	1	7	0	0.2	1	0.8	0.5	S_Chapter 16. Definite Assignment ; T_Branching Statements ; S_Chapter 14.11. The switch Statement
2: XML string parsing in Java	1	1	1	0.9	4	0.4	0.3	A_javax.xml.parsers.DocumentBuilder.parse() ; A_javax.xml.bind.DatatypeConverter ; A_javax.xml.transform.Transformer
3: PLAF can't change button color	2	1	0.4	0.5	1	0.4	0.3	A_javax.swing.UIManager ; T_How to Set the Look and Feel ; T_How to Use Color Choosers
4: match generics with mockito	1	1	0.4	0.5	1	0.6	0.7	T_Type Erasure ; T_Erasure of Generic Types ; T_Erasure of Generic Methods
5: reinitialise transient variable	1	-	0	0	1	0.6	0.7	A_java.io.Serializable ; S_Chapter 8.3.2. Initialization of Fields ; S_Chapter 14.14. The for Statement
6: write and read multiple byte[] in file	1	2	0.2	0.2	1	0.6	0.5	A_java.io.InputStream.read() ; A_java.io.OutputStream.write() ; A_java.io.RandomAccessFile
7: equality of boxed boolean	1	1	0.4	0.5	1	0.6	0.5	S_Chapter 5.1.7. Boxing Conversion ; A_java.util.Arrays.equals() ; A_java.lang.Object.equals()
8: resetting and copying two dimensional arrays	1	-	0	0	1	0.4	0.4	A_java.lang.System.arraycopy() ; T_Arrays ; A_java.util.Arrays.copyOf()
9: java standard on result of casting a double to an int	2	1	0.8	0.6	1	0.8	0.7	S_Chapter 5.1.3. Narrowing Primitive Conversion ; A_java.lang.Number ; A_java.lang.Double
10: registering and using a custom java.net.URL protocol	5	3	0.2	0.1	1	0.6	0.8	A_java.net.URLConnection ; A_java.net.URL ; A_java.net.HttpURLConnection
11: why is the protected method not visible	1	1	0.2	0.1	3	0.4	0.5	S_Chapter 5.1.3. Narrowing Primitive Conversion ; S_Chapter 5.1.2. Widening Primitive Conversion ; T_Controlling Access to Members of a Class
12: java date formatting ParseException	1	1	0.8	0.6	1	0.6	0.7	A_java.text.SimpleDateFormat ; A_java.text.DateFormat.parse() ; A_java.text.DateFormat.format()
13: java executor with no ability to queue tasks	2	-	0	0	1	0.8	0.8	A_java.util.concurrent.Executors ; A_java.util.concurrent.ThreadPoolExecutor
14: how to replace a JPanel based on user clicks	1	2	0.6	0.5	2	0.4	0.6	T_How to Use CardLayout ; T_How to Use the Focus Subsystem ; T_How to Write a Mouse Listener
15: ambiguous varargs method call compilation error	1	-	0	0	2	0.4	0.5	S_Chapter 15.12.2. Compile-Time Step 2: Determine Method Signature ; A_java.lang.invoke.MethodHandle ; S_Chapter 6.6.1. Determining Accessibility
16: why is there no generic type information at run time	2	1	0.8	0.7	1	0.6	0.6	T_Generic Methods ; T_Why Use Generics? ; T_Erasure of Generic Types
17: raster format exception (Y+height)	1	4	0.2	0.3	1	0.4	0.4	A_java.awt.image.BufferedImage ; A_javax.imageio.ImageIO ; T_Lesson: Working with Images
18: rendering combo boxes in a JTable	2	1	0.6	0.6	1	0.8	0.7	T_How to Use Tables ; T_How to Write an Item Listener ; T_How to Use Combo Boxes
19: java modifying a class directly, null reference	1	-	0	0	1	0.6	0.5	T_Anonymous Classes ; S_Chapter 12.4. Initialization of Classes and Interfaces ; A_java.lang.NullPointerException
20: windows azure date format to java date	1	-	0	0	1	0.8	0.7	A_java.text.SimpleDateFormat ; A_java.time.format.DateTimeFormatter ; A_java.text.DateFormat
21: reading arraylist from a .txt file	1	5	0	0.1	2	0.6	0.6	A_java.nio.file.Files.readAllLines() ; A_java.io.InputStream ; A_java.util.Scanner.nextLine()
22: close connection and statement finally	1	1	0.8	0.4	1	0.8	0.6	T_The try-with-resources Statement ; A_java.sql.Statement ; T_Using Transactions
23: when are java temporary files deleted	2	1	0.4	0.2	1	0.4	0.5	A_java.io.File.createTempFile() ; A_java.io.File.deleteOnExit() ; A_java.nio.file.Files.createTempDirectory()
24: shutdown application gracefully upon power loss	1	-	0	0	1	0.6	0.6	A_java.lang.Runtime.addShutdownHook() ; A_java.util.concurrent.ExecutorService.shutdownNow()
25: get single bytes from multi-byte variable	1	1	0.2	0.1	1	0.4	0.5	A_java.util.Timer
average	1.4	> 1.94	0.32	0.284	1.32†	0.576†	0.568†	T_Primitive Data Types ; A_java.nio.ByteBuffer ; T_Variables

we focus on official documentation in this study, we restrict the retrieved results by Google by set the ‘site’ parameter in the search. For example, the second query in Table 5.5 to Google is “XML string parsing in Java site:docs.oracle.com/javase” using Google search engine in August, 2016. Note that, all the three types of Java documentation (language specification, API documentation, and tutorial) are under the same site: docs.oracle.com/javase.

To measure the performance of QDLinker and Google, we use three metrics [Gu et al., 2016, Raghothaman et al., 2016]. *FR* is the rank of the first relevant result, as most users scan the results from top to bottom. The smaller the number of *FR*, the better the performance. The *P@5* denotes the precision of the top 5 ranked results. Note that, the relevance of the results are manually labeled by our annotators. Similarly, the *P@10* is the precision of the top 10 ranked results.

We recruited two developers to manually annotate the two set of results from Google and QDLinker, respectively. Each link to software documentation in result list was marked relevant or irrelevant, indicating whether the developer considered this software document is relevant to the query question. The annotation was done individually by the two developers and for inconsistent judgments, the two developers reached a consensus through discussion.

5.5.2 Evaluation Results

Table 5.5 shows the performance comparison of Google search and QDLinker. In particular, the symbol “-” in the second column indicates that there is no relevant software documentation returned by Google search in the query. The last row shows the average performance on the three metrics.

Compared with Google search, QDLinker achieves better performance on FR , $P@5$ and $P@10$. In most cases (20 out of the 25 queries), QDLinker is able to recommend relevant software documentation at the first position in the result list. The differences between these two approaches in terms of three metrics are statistically significant at $p < 0.05$. That is, QDLinker provides more relevant software documentation in top 10 results than Google search in our user study.

The last column shows web page titles of the top 3 ranked relevant documents by QDLinker, which consist of Java API documentation (marked by A_), Java language specifications (marked by S_) and Java tutorials (marked by T_). For example, in Queries 1 and 2, “S_Chapter 16. Definite Assignment”⁶, “A_javax.xml.parsers. DocumentBuilder. parse()”⁷, “T_Branching Statements”⁸ represent a document in language specification, API documentation, and tutorial, respectively. Compared with Google search, we make following observations:

- QDLinker can bridge the semantic gap between question and software documentation. For example, query 5 is “reinitialise transient variable”, and there is no any Java documentation which contains all the three keywords. Google search cannot return relevant results in this query. Likewise, the state-of-the-art API usage

⁶<https://docs.oracle.com/javase/specs/jls/se8/html/jls-16.html>

⁷<https://docs.oracle.com/javase/8/docs/api/javax/xml/parsers/DocumentBuilder.html#parse-java.io.File->

⁸<https://docs.oracle.com/javase/tutorial/java/nutsandbolts/branch.html>

miner [Gu et al., 2016] cannot return any API sequences based on code corpus from Github. It definitely alerts users with the prompt of “*Note: your query may not be supported by Java SDK library*”. We manually check our training dataset and find that some community users have implemented the task using the class “`java.io.Serializable`” and method “`readObject`” in Stack Overflow. Thus, QDLinker can effectively answer this question because it takes into account content and context of software documentation simultaneously.

- QDLinker can effectively answer complex and bug-like queries. For instance, query 17 “*raster format exception (Y+height)*” and query 19 “*java modifying a class directly, null reference*” are related to program exceptions. Poorer results were obtained from Google for such kind of queries. On the contrary, QDLinker provided high quality results for such kind of queries, and an example is the API documentation `java.awt.image.BufferedImage` for query 17.
- QDLinker can effectively answer programming questions which are in specific usage scenarios. For instance, query 18 “*rendering combo boxes in a JTable*” is about usage of *combo boxes* in the scenario of “*JTable*” and query 13 “*java executor with no ability to queue tasks*” is about of *Java executor* in the scenario of “*queue tasks*”. The official software documentation does not serve as good reference for these queries in specific usage scenarios, while QDLinker can provide high quality software documentation for these queries. For example, the API `java.util.concurrent.ThreadPoolExecutor` is a high-quality API document for query 13.

5.6 Summary

Developers often encounter questions in specific programming tasks. Although programming languages and software packages are well supported by formal documentation, the documentation aims at comprehensive coverage and not on specific tasks. The semantic gap between the developers’ questions and software documentation make it difficult for developers to search for the most relevant documentation. Utilizing the social context available at Stack Overflow, we built QDLinker to bridge the gap

between the questions and documentation. Given a programming question, QDLinker returns the links to the most relevant documentation. The semantic features between questions and software documentation in QDLinker are learned through a four-layer deep neural network. Together with content features, the learned features are fed to a learning-to-rank schema for ranking the most relevant software documentation. Using real questions from Stack Overflow, we show that QDLinker effectively locates the most relevant software documentation to questions, and its performance significantly outperforms baseline methods.

The proposed QDLinker framework may benefit other software engineering problems. First, considering the ability of bridging the semantic gap between programming questions and software documentation, QDLinker could improve official software documentation with the information in questions and answers. Second, current code search does not support natural language. QDLinker can be integrated in code search engines to improve code search performance. Third, this work opens several interesting directions for future work with regard to automatic conversation between humans and computers. In the future, we will explore the applications of QDLinker to these problems.

Distilling Crowdsourced API Negative Caveats from Web Q&A Discussions to Augment Software Documentation

All things are difficult before they are easy.

— Thomas Fuller

Negative caveats of API are about “how not to use an API”, which are often absent from the official API documentation. When these caveats are overlooked, programming errors may emerge from misusing APIs, leading to heavy discussions on Q&A websites like Stack Overflow. If the overlooked caveats could be mined from these discussions, they would be beneficial for programmers to avoid misuse of APIs. However, it is challenging because of the discussions are informal, redundant, and diverse. In this chapter, we propose **DISCA**, a novel approach to automatically **DIS**tilling desirable API negative **CA**veats from unstructured Q&A discussions. Through sentence selection and prominent term clustering, **DISCA** ensures the distilled caveats are *context-independent, prominent, semantically diverse and non-redundant*. Quantitative evaluation in our experiments shows that the proposed **DISCA** significantly outperforms four text-summarization techniques. We also show that the distilled API negative caveats could greatly augment API documentation through qualitative analysis.

6.1 Background and Motivation

Application Programming Interfaces (APIs) are foundations of software development. To program to an API, developers need to know not only “how to use the API”, but also “how **not** to use the API”. Table 6.1 lists 7 examples of “how not to use an API” extracted from Stack Overflow, a Q&A website for topics in programming. We refer to such “how not to use an API” directives as *API negative caveats*.

API documentation is an important resource for developers to learn unfamiliar APIs [Kramer, 1999, Robillard and Deline, 2011, Dagenais and Robillard, 2012, Stylos

Table 6.1: Example API negative caveats extracted from Stack Overflow

API Types	API negative caveats	Post ID
java.util.HashMap	Don't use a HashMap if you are going to have multiple threads, use a ConcurrentHashMap instead.	15389917
	HashMap doesn't guarantee the order in which elements are returned.	10372667
javax.swing.JTextArea	The JTextArea will not scroll down with the text. JTextArea is not a component designed for styled text.	630030 9097995
java.awt.event.ActionListener	Inner classes, such as your ActionListener, cannot access non-final variables from the scope that contains it.	30689818
	Don't use an ActionListener to check when the button is selected.	19532759
javax.swing.text.html.ListView	You cannot use a ListView inside a ScrollView (any two views that have same orientation scrolling).	8551849

et al., 2009]. By providing important information about functionality, parameters and usage scenarios of an API, API documentation often does a good job at explaining “how to use an API” [Robillard and Deline, 2011, Subramanian et al., 2014]. More often than not, API documentation does not mention any API negative caveats. Even when negative caveats are mentioned, they are often buried in the verbose descriptions of the API and can be barely noticed by developers. For example, Figure 6.1 shows the screenshots of three kinds of typical API documentation.

Not mentioning API negative caveats is not always because API designers are reluctant to document negative caveats. We will use the examples in Table 6.1 to further illustrate this point. First, an API negative caveat is sometimes related to a broader context in which an API is used. For example, java.awt.event.ActionListener is often implemented as an inner class. According to Java language specification, an inner class cannot access non-final variables from the scope that contains the inner class. Second, an API negative caveat may be rooted in the API’s design. For example, javax.swing.JTextArea is designed to display plain text only; as such, it does not support styled text. Third, API negative caveats may also emerge from practical usage scenarios, which API designers are unable to foresee. As an example, it is not expected that javax.swing.text.html.ListView to be used inside a ScrollView.

Nevertheless, once an API negative caveat has slipped programmer’s attention, it is very likely to result in unexpected programming errors. An effective way of seeking solution is to post a question on Stack Overflow, and wait for suggestions from


```
public class JTextArea
  extends JTextComponent
  ...omit 8 lines...
```

The `java.awt.TextArea` internally handles scrolling. `JTextArea` is different in that it doesn't manage scrolling, but implements the swing `Scrollable` interface. This allows it to be placed inside a `JScrollPane` if scrolling behavior is desired, and used directly if scrolling is not desired.

The `java.awt.TextArea` has the ability to do line wrapping. This was controlled by the horizontal scrolling policy. **Since scrolling is not done by `JTextArea` directly, backward compatibility must be provided another way.** `JTextArea` has a bound property for line wrapping that controls whether or not it will wrap lines. By default, the line wrapping property is set to false (not wrapped).
...

```
public interface ActionListener
  extends EventListener
```

The listener interface for receiving action events. The class that is interested in processing an action event implements this interface, and the object created with that class is registered with a component, using the component's `addActionListener` method. When the action event occurs, that object's `actionPerformed` method is invoked.

```
public class ListView
  extends BlockView
```

A view implementation to display an html list

Figure 6.1: Examples of three official API documents

other developers. Often, the answers explicitly point out the overlooked API negative caveats and suggest ways to avoid such errors as shown in Table 6.1.

Such Q&A discussions effectively document negative experiences emerging from overlooking API negative caveats in practice. They are also referred to as crowd documentation [Parnin et al., 2012], which generates a rich source of content that complements the official API documentation. If we could extract such crowdsourced API negative caveats like those in Table 6.1, we may highlight the hard-to-notice negative caveats in API documentation or augment the API documentation with the missing negative caveats. Such augmentation would raise developer's caution to avoid misuse of APIs, or to help them fix errors caused by overlooking API negative caveats. However, these API negative caveats, present in crowdsourced Q&A discussions, are

informal, redundant, and are often related to different aspects of API usage. How to effectively distill API negative caveats from unstructured Q&A discussions is a challenging task.

In this chapter, we present **DISCA**, an approach to automatically distilling API negative caveats from large-scale unstructured Q&A discussions. To the best of our knowledge, our work is the first attempt to tackle the problem about negative usages of APIs. We formulate the problem as a text-summarization task and identify four desirable properties for the distilled API negative caveats: *context-independence*, *prominence*, *semantic diversity*, and *semantic non-redundancy*.

Given a set of programming-related sentences extracted from Stack Overflow discussions, **DISCA** first selects a set of candidate sentences, *i.e.*, sentences mentioning a specific API with negative expressions. Then, **DISCA** selects *context-independent* sentences that identify issues about an API usage without referring to the discussion contexts. Next, **DISCA** selects *semantically diverse* and *non-redundant* sentences that cover *prominent* domain-specific terms through a combination of techniques including relative entropy, term co-occurrence analysis, and set cover.

We conduct both quantitative and qualitative evaluations to demonstrate the effectiveness of **DISCA**. For quantitative evaluation, we compare the performance of **DISCA** against four text-summarization techniques as baselines: eigenvector centrality of sentence graph (LexRank [Erkan and Radev, 2004]), topic modeling (LDA [Blei et al., 2003]), sentence clustering (KM [MacQueen et al., 1967]), and sentence diversification (MMR [Goldstein et al., 1999]). Following the common procedure of evaluating text-summarization techniques [Wang et al., 2012, Ganesan et al., 2010], we recruit three developers to generate gold standard summaries of API negative caveats for 10 Java APIs that have different mention frequencies on Stack Overflow. We evaluate the performance of **DISCA** and baselines with two commonly used metrics: Recall-Oriented Understudy for Gisting Evaluation (ROUGE) and Normalized Discounted Cumulative Gain (nDCG). Our results show that **DISCA** outperforms the four baselines by 10.60% to 22.47% for ROUGE and 17.63% to 42.87% for nDCG, respectively. Our qualitative evaluation compares the negative caveats that are documented in the API documentation of these 10 Java API types and the ones mined by **DISCA**. Results

show that official API documentations mention only 6 negative caveats, while **DISCA** distills 164 from Stack Overflow. These 164 negative caveats cover 4 out of the 6 negative caveats mentioned in API documentation (*i.e.*, recall=67%). More importantly, **DISCA** greatly augments the official API documentations of the 10 Java APIs with 146 correct negative caveats (*i.e.*, precision=89%).

6.2 Problem Definition

The raw input to our approach is a set of programming-related sentences extracted from online discussion. Such sentences can be easily obtained from Q&A websites such as Stack Overflow. Given an API type, *e.g.*, a class or an interface declared in Java SDK like `java.util.HashMap`, our task is to distill a small set of sentences as negative caveats related to the concerned API. We formulate the problem of distilling desirable API negative caveats as a text-summarization task, through the following three definitions.

Definition 6.10 (Candidate sentences) *Let S_{raw} be a set of programming-related sentences from online discussion, and let x be an API type. A candidate sentence for API type x is a sentence $s \in S_{raw}$ that mentions API x and contains negative expression(s). We denote the set of candidate sentences for API x as S_x .*

Definition 6.11 (Candidate API negative caveats) *Candidate API negative caveats is a set of sentences, denoted by $Cand_x \subseteq S_x$, after removing context-dependent sentences from candidate sentences.*

Definition 6.12 (Desirable API negative caveats) *Given an API type x , a set of desirable API negative caveats is a small subset of sentences, denoted by $\mathcal{A}_x \subseteq Cand_x$. $\mathcal{A}_x$ represents the semantically diverse and non-redundant sentences that cover the most prominent domain-specific terms related to the usage issues of API type x .*

Next, we detail the four desired properties: context-independence, prominence, semantic diversity, and non-redundancy.

Context-independence. Sentences in a discussion often reference to other part(s) of the discussion, for example, “*The following article addresses this question in some details about HashMap*”. We consider such sentences context-dependent. The distilled API negative caveats should be context-independent, so that programmers know a negative caveat without having to refer to the original discussion. Having said that, the property of context-independence does not mean that programmers do not require any additional knowledge about the API to fully understand its negative caveats.

Prominence. Discussion about an API may cover many different aspects, not limited to negative caveats. Consider two sentences “*HashMap essentially has $O(1)$ performance*” and “*HashMap is not synchronized*”. The first sentence is about time complexity while the second sentence is our main focus. More specifically, an API negative caveat is usually concerned about some domain-specific terms related to the API usage, for instance *multi-thread*, *synchronization*, *thread-safe*, *sort* for `java.util.HashMap`. Identifying such prominent domain-specific terms is crucial for important API usage issues.

Semantic diversity. An API type often has negative caveats related to different ways of using the API. For example, `java.util.HashMap` does not support *multi-thread* and does not guarantee *element order*. The number of sentences on different aspects often varies greatly in discussion. This calls for a proper way of handling data imbalance to avoid getting sentences all for one aspect of an API type. Distilling semantically diverse sentences reveals a more complete picture of an API’s negative caveats.

Semantic non-redundancy. In a large volume of informal discussion, the same API negative caveat may be mentioned many times but in different wording. “*A HashMap does not maintain an order*” and “*This is the property of HashMap where elements are not iterated in the same order in which they were inserted*” express similar meaning but have low lexical similarity. In such cases, we would like to select sentences that convey richer information about the API usage, e.g., the second sentence in this example. At the same time, we should avoid selecting other sentences which are semantically redundant.

Figure 6.2: Overview of the DISCA framework

6.3 The Disca Approach

To solve the text-summarization problem as defined above, we propose **DISCA** (for **DI**stilling crowdsourced API negative **CA**veats), as shown in Figure 6.2.

6.3.1 Input Data

The input data to **DISCA** includes: a set of APIs identified by their full qualified names and a set of sentences from user discussions. In this work, we focus on classes and interfaces defined in Java SDK as the API types of interest.¹ Sentences are extracted from Stack Overflow, considering its popularity among programmers and the volume of the data. More specifically, sentences are extracted from Stack Overflow posts that are tagged with Java. We preserve the textual content by removing HTML tags, and we remove long code snippets enclosed in `< pre >``< code >` but keep short code elements in `< code >`. To help to determine the quality of the sentences, we attach votes to sentences based on the number of votes received by its original post.

Based on our observations on Stack Overflow, we remove three types of sentences that are unlikely to discuss API negative caveats. First, the sentences in question are removed, for example *“ListView adapter with HashMap isn’t displaying correctly”*, because these sentences are more likely to discuss programming problems rather than the cause of the problems. Second, the interrogative sentences are removed, for example *“Have you overridden the keySet() method in your HashMap?”*; these sentences pose questions rather than give solutions. Third, we remove opinion-based sentences with subjective opinions, such as *“I’m not sure ...”*, *“I do not think ...”*, etc.

¹<https://docs.oracle.com/javase/8/docs/api/overview-summary.html>

6.3.2 Selecting Candidate Sentences

Given an API type and a set of sentences, the first step of our approach is to select a set of *negative* sentences that *mention* the API type. The selected sentences are candidate sentences from which API negative caveats will be distilled.

Selecting sentences that mention an API. Because we want to distill API negative caveats of an API, the candidate sentences should mention the given API. This task is often referred to named entity recognition [Tjong Kim Sang and De Meulder, 2003] and entity linking [Shen et al., 2015]. As entity recognition and linking are not the key focus of this work, we adopt a name-matching strategy to select sentences that mention a given API. More specifically, we use a software-specific tokenizer [Ye et al., 2016a] to tokenize the sentences. This tokenizer preserves the integrity of code-like tokens like `java.util.HashMap` and the sentence structure. If a token in a sentence matches the full or partial name of an API, the sentence is considered mentioning the API. Although this strategy is simple, it has shown effectiveness (precision 0.92 and recall 0.97) in several studies [Treude and Robillard, , Ye et al., 2016b, Bacchelli et al., 2011, Rahman et al., 2016] for recognizing mentions of APIs with distinct orthographic features.

When selecting candidate sentences, variations of API mentions have to be taken into account. Informal discussions on social platforms (*e.g.*, Stack Overflow) are contributed by millions of users with diverse technical and linguistic background [Ye et al., 2016b, Chen et al., 2017]. Such informal discussions are full of misspellings and synonyms [Beyer and Pinzger, 2016, Ye et al., 2016b, Chen et al., 2017]. Consequently, the same API is often mentioned in many different forms intentionally or accidentally. Example mentions of “HashMap” include “hash map”, “hashmaps” and “hash-map”. We resort to the software-specific synonym thesaurus [Chen et al., 2017] to match API-mention variations. This synonym thesaurus documents commonly seen misspellings and synonyms mined from Stack Overflow.

Selecting sentences with negative expressions. API negative caveats are usually expressed in negative sentences, *i.e.*, sentences containing negative expressions. To this end, we use dependency parse tree to detect negative sentences. Dependency parse

Figure 6.3: Three examples of dependency parsing for sentences: “HashMap doesn’t define the order of iteration over the elements”, “Don’t use a HashMap with multiple threads” and “JSONObject does not have too much additional overhead on top of a HashMap”. Note that the two examples on the left hand side are selected as candidate sentences in this study.

tree provides a representation of grammatical relations between words in a sentence.² It is a directed graph where nodes represent words and edges represent syntactic roles, *e.g.*, *nsubj*: nominal subject, *aux*: auxiliary, *det*: determiner, etc. Among these syntactic roles, we can use negation modifier (*i.e.*, *neg*) to detect negative expressions. Figure 6.3 illustrates two examples dependency parse trees produced by Stanford Parser³. Syntactic roles of negation, *i.e.*, *neg(define, n’t)* and *neg(use, n’t)* can be detected from these two examples (highlighted in orange in Figure 6.3).

To ensure the negative expressions are on APIs, we select only negative sentences whose subject or object is a given API. For example, both sentences “JSONObject does not have too much additional overhead on top of a HashMap” and “HashMap doesn’t define the order of iteration over the elements” are negative sentences and both mention HashMap. Only the second sentence is selected as a candidate sentence for HashMap because the negative expression is on the API. More specifically, a given API must exist in *nsubj* or *dobj* syntactic role in a sentence, as highlighted in blue in Figure 6.3.

6.3.3 Filtering Out Context-Dependent Sentences

Context-dependent sentences are less meaningful without referring to the original discussion where the sentences appear. We remove context-dependent sentences from the candidate sentences, based on a set of predefined sentence patterns.⁴ The patterns

²<http://universaldependencies.org/en/dep/all.html>

³<http://nlp.stanford.edu/software/lex-parser.html>

⁴See the full list of defined patterns at <http://128.199.241.136/disca/appendix> or A.2

are defined from our observation made on the data. The first category of patterns removes sentences that reference to code snippets in the discussion, such as “*An equivalently synchronized HashMap can be obtained by ... some code ...*”. The second category removes sentences that reference to demonstrative pronoun (e.g., “do this”, “like this”, “this won’t”, etc.), for example, “*If you are trying to do this in a single thread, I would recommend HashMap*”. The third category removes sentences that reference to another sentence in the discussion (e.g., “see the next step”, “the following”, etc.), for example, “*The following article addresses this question in some detail: HashMap requires a better hashCode()*”. We refer to the rest of context-independent sentences as *candidate API negative caveats*, denoted by $Cand_x$.

6.3.4 Identifying Prominent Domain-specific Terms

An API negative caveat is usually concerned about domain-specific terms related to the particular API usage. Identifying prominent terms in candidate API negative caveats helps to distill frequently-overlooked but important API usage issues.

Inspired by Park *et al.* [Park et al., 2008] and Chen *et al.* [Chen et al., 2017], we identify prominent terms by contrasting term frequency of a term in candidate API negative caveats and its frequency in background corpus. Recall that $Cand_x$ represents the set of candidate API negative caveats for API type x . For the sentences in $Cand_x$, we build a term (unigram) vocabulary V_x after removing stop words and performing word stemming. For a term $t \in V_x$, we use relative entropy to weight its prominence: $w(t) = p(t) \log \frac{p(t)}{q(t)}$, where $p(t)$ is the probability of observing t in $Cand_x$ and $q(t)$ is probability of observing t in all Stack Overflow posts that are tagged with the corresponding programming language, *i.e.*, Java in our setting. Based on the term weight, we select the top- k ($k=100$ in this work) ranked terms as the prominent domain-specific terms in candidate API negative caveats. Note that, the setting of k may not significantly affect the results as the prominent terms will be grouped to semantic aspects, to be discussed next.

6.3.5 Discovering Semantically Diverse Aspects

A group of semantically-related terms together reveal a semantic aspect of API usages, for example, (*thread, synchronization, safe*) for the issue of using `java.util.HashMap` in

multi-thread settings, (*key*, *hashcode*, *equal*) for the element uniqueness issue of `java.util.HashMap`, and (*order*, *insert*, *iterate*) for the element ordering issue of `java.util.HashMap`. Clustering semantically-related prominent terms helps to discover semantic aspects of an API, which in turn help to distill semantically diverse API negative caveats.

Semantic relatedness between terms can be discovered from term co-occurrence in sentences [Hua et al., 2015, Lund and Burgess, 1996]. To capture semantic relatedness between all prominent terms, we construct a term co-occurrence graph, where nodes are prominent terms and edges reflect the frequencies of term co-occurrences in candidate API negative caveats. An edge is added between two terms if their co-occurrence frequency is above a threshold. To discover the different aspects about an API, we cluster prominent terms in the graph into a set of disjoint term communities. In particular, we use Louvain method [Blondel et al., 2008]. It iteratively optimizes local communities until global modularity no longer improves. Figure 6.4 shows the community detection results for prominent terms of API `java.util.HashMap` in our evaluation. This graph is constructed from the top-100 prominent terms in the candidate API negative caveats, and the edge co-occurrence frequency threshold is set to 3. Observe that the detected term communities are semantically diverse (highlighted in different colors in Figure 6.4). Each term community represents one key semantic aspect of `java.util.HashMap`, including comparator implementation, element order, key/hashcode, and multiple threads.

6.3.6 Selecting API Negative Caveats

The final step of **DISCA** is to select sentences to represent each term community discovered in the earlier step. We select sentences based on three intuitions: (1) prominence: the selected sentences should be as prominent as possible; (2) quality: sentences should be of high quality, preferred from highly voted answer posts; (3) redundancy: the selected sentences should minimize redundant information.

Based on the three intuitions, we formulate the selection of desirable API negative caveats as a weighted *set cover* problem. Given an API type x , let $T_x = \{t_1, t_2, \dots, t_N\}$ be a set of N prominent terms in its term co-occurrence graph. Assume M term-communities are detected in the term co-occurrence graph, i.e., $\Phi_x = \{C_1, C_2, \dots, C_M\}$

Figure 6.4: Term Communities shown in different colors identified from the term co-occurrence graph of `java.util.HashMap`

($C_i \cap C_j = \emptyset$). For a term-community $C_m \in \Phi_x$, there are K prominent terms in this community, i.e., $C_m = \{t_1^m, t_2^m, \dots, t_K^m\}$, where the superscript m indicates the m -th community, $t_K^m \in T_x$ and $\bigcup C_m = T_x$. Recall that $Cand_x$ is a set of candidate API negative caveats for API type x . For each sentence $s_i \in Cand_x$, we represent s_i as a set of prominent terms in the sentence, i.e., $s_i = \{t_1^i, t_2^i, \dots, t_W^i\}$ where the superscript i indicates the i -th sentence and $t_W^i \in T_x$. The two sentences in $Cand_x$ may have overlapping terms. Each sentence s_i has a $cost(s_i)$. The goal is to find a set cover $\mathcal{A}_x \subseteq Cand_x$ of minimal total sentence weight to cover all terms in T_x , i.e.,

$$\begin{aligned}
& \text{Minimize } \sum_{s_i \in \mathcal{A}_x} \text{cost}(s_i) \\
& \text{Subject to } \bigcup_{s_i \in \mathcal{A}_x} s_i = T_x
\end{aligned} \tag{6.1}$$

The $\text{cost}(s_i)$ is computed from two parts: (i) average value of the prominence score of terms in sentence s_i by $w(t)$ defined in Section 6.3.4, denoted by $\text{prom}(s_i)$, and (ii) post score $\text{post}(s_i)$, the user votes of the post that contains sentence s_i . These two scores are normalized independently based on their corresponding maximum and minimal values. Then $\text{cost}(s_i)$ is a linear combination of the two scores.

$$\text{cost}(s_i) = -\alpha \cdot \text{prom}(s_i) - \beta \cdot \text{post}(s_i) \tag{6.2}$$

where α and β are the coefficients and $\alpha + \beta = 1$. The intuitive interpretation of Equation 6.2 is that the selected sentences should be as prominent as possible and have been up-voted by many users. The two coefficients control the trade-off between prominence and quality of sentences (set equal in this work).

The weighted set cover problem in Equation 6.1 is NP-hard [Aho and Hopcroft, 1974]. But there is a polynomial time greedy approximate algorithm, which provides a $O(\log n)$ approximate solution [Blondel et al., 2008]. Algorithm 3 shows the steps of this greedy approximate algorithm for selecting a set of representative sentences from candidate API negative caveats to satisfy Equation 6.1. Instead of selecting sentences to cover the term set T_x as a whole, we use a divide-and-conquer strategy that selects sentences for one randomly selected term community at a time (Lines 2-3). This divide-and-conquer strategy, together with non-redundant sentence selection mechanism (Line 7), ensures the semantic diversity of the selected sentences, even though the mentions of API negative caveats related to different semantic aspects of an API are imbalanced.

For a term community C_m , the inner loop (Lines 4-10) continues until the union of prominent terms in the selected sentences ψ_m , covers all terms in C_m . The notation $|s_i \setminus T(\psi_m)|$ (Line 7) denotes the number of terms in s_i but are not in the selected sentences ψ_m . The notation $p(s_i)$ is the production of $\text{cost}(s_i)$ and the number of newly

Algorithm 3: Greedy Selection Algorithm

Input: candidate API negative caveats $Cand_x$; prominent terms T_x ; term communities Φ_x

Output: $\mathcal{A}_x$: Map of API negative caveats for term communities

```

1  $\mathcal{A}_x = \emptyset$ ;
2 foreach  $C_m$  in  $\Phi_x$  do                                     // iterate through communities
3      $\psi_m = \emptyset$ ;                                           // selected sentences for  $C_m$ 
4     while  $\psi_m \neq C_m$  do
5         foreach  $s_i$  in  $Cand_x$  and  $s_i \cap C_m \neq \emptyset$  do
6             Compute  $cost(s_i)$  by Equation 6.2;
7             Compute  $p(s_i) = cost(s_i) \cdot |s_i \setminus \psi_m|$ ;
8             Select  $s_j \in Cand_x$  where  $p(s_j)$  has the minimal value;
9              $\psi_m \leftarrow \psi_m \cup \{s_j\}$ ;
10             $Cand_x \leftarrow Cand_x \setminus \{s_j\}$ ;                // remove  $s_j$  from  $Cand_x$ 
11         $\mathcal{A}_x \leftarrow \mathcal{A}_x \cup \{map\langle C_m, \psi_m \rangle\}$ ;        // map⟨key, value⟩
12 return  $\mathcal{A}_x$ 

```

added terms if s_i is selected (Line 7). The intuition is that the more new terms bring in by selecting sentence s_i , the more likely the sentence will be selected. A sentence once selected is removed from candidate negative caveats set $Cand_x$ (Line 10). The algorithm returns the map of the selected sentences for each term community as desirable API negative caveats for API x .

Consider the “multiple thread synchronized” term community for `java.util.HashMap` (in orange color) in Figure 6.4. Figure 6.5 shows the API negative caveats selected by Algorithm 3 for this term community. Observe that all three negative caveats are related to *multi thread* and *synchronization* issues of `HashMap`, and there is no redundancy. Instead, the selected sentences together provide complementary information for better understanding the issues, compared to the sentence “this implementation is not synchronized” in the official API documentation of `HashMap`. The second and third sentences even provide alternative APIs that are not mentioned in the official documentation.

6.4 Evaluation

In this section, we first detail the experimental settings, then present quantitative and qualitative evaluation.

Figure 6.5: Selected API negative caveats for the term community “multiple thread synchronized”

6.4.1 Experimental Settings

Data Collection. From the official Java 8.0 website, we obtained 4,240 Java API types. We collect all Stack Overflow posts tagged with Java from the March-2016 data dump as the general corpus. Among the posts, 1,081,439 sentences mention at least one Java API type. For the top 10 most frequently mentioned Java packages, we choose the top 1 frequently-mentioned API type in each package in our evaluation. Reported in Table 6.2 (the first three column), the 10 API types have a wide range of mention frequency (MF) and candidate API negative caveats ($Cand_x$) ranging from tens to hundreds of sentences.

Parameter setting for DISCA. The configuration of DISCA is based on the performance of a development set (java.util.HashMap). Accordingly, for each of the 10 API types, we use the top-100 prominent terms in its candidate API negative caveats to construct the term co-occurrence graph with term co-occurrence frequency set at 3. The resulting term co-occurrence graph has 3 to 6 term-communities. The α and β parameters for $cost(s_i)$ are set to 0.5.

Baseline methods. We compare DISCA with four text-summarization methods. All methods take candidate API negative caveats for an API type as input, and independently select a subset of sentences as summaries.

Table 6.2: The statistics of Java API types used in evaluations. “ $Cand_x$ ” and “DISCA” denote number of candidate API negative caveats and number of negative caveats mined by DISCA, respectively

API Types	Mention frequency $Cand_x$ # Term communities DISCA			
java.util.ArrayList	55,802	689	4	18
javax.swing.JFrame	27,468	302	6	17
java.lang.NullPointerException	20,079	133	4	24
javax.xml.bind.JAXB	14,445	191	6	16
java.io.IOException	7,223	97	4	18
java.awt.event.ActionListener	7,014	82	5	17
java.sql.ResultSet	6,948	94	6	14
java.text.SimpleDateFormat	6,585	136	5	22
java.math.BigDecimal	6,568	78	3	12
java.nio.ByteBuffer	3,139	26	3	6

- **LexRank:** This method selects important sentences based on the concept of eigenvector centrality in a graph representation of sentences [Erkan and Radev, 2004]. Cosine similarity is used to calculate the similarity between two sentences.
- **LDA:** This method represents each sentence in a vector space using Latent Dirichlet Allocation (LDA) topic model [Blei et al., 2003]. For each topic, the sentence with the maximum probability is selected as an API negative caveat.
- **KM:** This method represents each sentence with a TF-IDF vector and performs k -means algorithm [MacQueen et al., 1967] to cluster the sentences, then chooses the centroids in clusters as API negative caveats.
- **MMR:** This method iteratively select API negative caveats with the maximal marginal relevance that measures novelty and diversity of the selected sentences [Goldstein et al., 1999].

Evaluation Metrics. We use two evaluation metrics, namely, ROUGE and nDCG.

- **ROUGE** measures the quality of a summary by counting the unit overlaps between a machine generated summary and a set of gold standard summaries.

ROUGE-N is the n -gram recall computed as follows:

$$ROUGE - N = \frac{\sum_{S \in ref} \sum_{gram_n \in S} Count_{match}(gram_n)}{\sum_{S \in ref} \sum_{gram_n \in S} Count(gram_n)} \quad (6.3)$$

where n represents the length of the n -gram, ref is the set of the gold standard summaries.

In our evaluation, we used ROUGE toolkit [Lin, 2004] (version 1.5.5) with ROUGE-1 (unigram-based) and ROUGE-2 (bigram-based). We also use ROUGE-SU4 that measures unigram recall and skip-bigram recall with maximum skip distance of 4. These three ROUGE measures have been shown to be able to identify machine generated summary that is the most correlated with human summaries [Lin and Hovy, 2003, Ganesan et al., 2010].

- **nDCG** measures the performance of a ranked list based on graded relevance levels.

$$nDCG@k = \frac{1}{IDCG} \sum_{i=1}^k \frac{2^{rel_i} - 1}{\log_2(i + 1)} \quad (6.4)$$

In this equation, $IDCG$ is the discounted cumulative gain of ideal ordering for all instances, rel_i is the relevance score of the i -th element in the ranked list. LexRank and MMR output a ranked list of sentences. For LDA and KM, we rank the centroids based on cluster size. In DISCA, Algorithm 3 ranks the API negative caveats for a term community. A ranked list of API negative caveats across communities is then obtained by ranking all term-communities based on the highest degree centrality of the communities. For the relevance judgment, the relevance level of each negative caveat is defined as the number of annotators who select the sentence. For example, the relevance of a negative caveat is 3 if all three annotators select this sentence in their gold standard summary and the relevance is 0 if no annotator selects it.

Gold standard generation. To make a fair comparison, selecting the same number of words or sentences is commonly used in the comparison of text-summarization methods [Wang et al., 2012, Ganesan et al., 2010]. In our experiments, for each API type in Table 6.2, we use each of the four baseline methods to select the same number of

Table 6.3: Agreement between annotators, by evaluating the summary of one annotator against the summaries of the other two annotators on three ROUGE measures

Annotator	ROUGE-1	ROUGE-2	ROUGE-SU4
Human A	0.7478	0.6169	0.6231
Human B	0.7609	0.6393	0.6410
Human C	0.7626	0.6414	0.6447
Average	0.7571 $\pm$ 0.0081	0.6325 $\pm$ 0.0136	0.6363 $\pm$ 0.0116

API negative caveats as **DISCA** selects. Then we mix the selected sentences of the five methods for human annotation. The annotators do not know which sentences are from which methods. We recruit three annotators who all have more than 4 years of programming experiences in Java and are familiar with the 10 Java types in Table 6.2. Because there are five methods, we ask each annotator to select 20% of sentences from the mixed sentences, based on the following criteria: (1) the selected sentences should cover prominent and diverse topics, and (2) the selected sentences should be informative and context-independent.

Overall, 25.9%, 19.2%, 14.5%, 20.3% and 20.1% of the selected sentences by annotators are from **DISCA**, LexRank, LDA, KM, and MMR, respectively. We consider the selected sentences by the three annotators as three independent gold standard summaries for an API type, because the evaluation metric ROUGE can handle multiple gold standard summaries.

Inter-Annotator Agreement. We use the Jackknifing procedure [Lin, 2004] to quantitatively assess the inter-annotator agreement. With this procedure, the ROUGE scores are computed over K sets of $K - 1$ reference summaries. That is, each human summary is evaluated against the remaining $K - 1$ gold standard summaries, and the average ROUGE scores are computed as reported in Table 6.3. Observe from the table that the average scores of ROUGE-1, ROUGE-2 and ROUGE-SU4 are 0.7571, 0.6325 and 0.6363, respectively. The largest standard deviation is about 0.0136, which indicates that the ROUGE scores of different annotators are close to the mean. In short, we can see that the annotators have a good agreement amongst themselves.

Table 6.4: Performance of the five methods on ROUGE measures. The best performance is highlighted in bold face. †indicates that the improvements made by **DISCA** over the best baseline is statistically significant under paired t -test with $p \leq 0.05$.

Method	ROUGE-1	ROUGE-2	ROUGE-SU4
LDA	0.5473	0.3202	0.3550
KM	0.6026	0.3498	0.3836
LexRank	0.6045	0.3555	0.3923
MMR	0.6097	0.3583	0.3868
DISCA	0.6269†	0.4152†	0.4374†

6.4.2 Quantitative Evaluation

Comparison against Gold Standard Summaries. We first report the ROUGE and nDCG scores for the negative caveats produced by each method against the gold standard summaries. Table 6.4 reports the ROUGE scores of the five methods, and Figure 6.6 plots nDCG values of these methods at different rank positions. We also list ROUGE-1, ROUGE-2 and ROUGE-SU4 for the 10 Java API types using **DISCA** in Table 6.5. From the results, we made the following observations.

First, **DISCA** achieves the best performance against the four baseline methods in terms of all ROUGE scores and all nDCG values. In terms of ROUGE scores, **DISCA** achieves 22.47%, 12.25%, 10.66% and 10.60% improvements over LDA, KM, LexRank, and MMR, respectively. For nDCG, **DISCA** achieves 42.87%, 17.65%, 20.01% and 17.63% improvements over the four methods. The improvements are statistically significant for the three kinds of ROUGE scores and nDCG under paired t -test with $p \leq 0.05$. We attribute this to the fact that **DISCA** takes context-independence, prominence, semantic diversity and semantic non-redundancy into account when selecting desirable API negative caveats.

Second, LDA yields the worst performance in terms of ROUGE scores and nDCG. A challenge in using LDA is to set an appropriate number of topics. In this evaluation, the number of topics is based on the number of API negative caveats that **DISCA** distills. From the results of LDA, we note that too many topics result in the extracted topics to be similar to each other. On the other hand, too few topics makes the extracted topics less meaningful or non-interpretable. Unfortunately, without prior

Figure 6.6: nDCG values of the five methods at different ranking positions {1, 2, 3, 4, 5, all}

knowledge of the distribution of candidate API negative caveats, it is difficult to set the right topic number. According to ROUGE scores in Table 6.4, MMR outperforms KM. That is, the summaries of MMR is closer to gold standard summaries than KM without considering the relevance ranking of selected sentences. However, when considering the relevance ranking of the selected sentences, Figure 6.6 shows that KM outperforms MMR in nDCG values. This is because the sentences selected by KM are ranked by cluster size which reflects the prominence of the selected sentences. The performance of LexRank is comparable to that of MMR, because LexRank takes into account both relevance ranking and diversity. Recall that, LexRank first ranks candidate API negative caveats in a graph model based on sentence similarity. Then it uses a greedy algorithm to select diverse sentences. The main issue of LexRank and MMR is that both are based on sentence-level lexical similarity, which cannot distinguish lexically different but semantically redundant sentences.

Third, although the performance of **DISCA** varies for different API types in Table 6.5, **DISCA** outperforms all baseline methods for all API types.⁵ Among the 10

⁵Detailed results not reported for the interests of page space.

Table 6.5: ROUGE scores obtained by **DISCA** on different Java API types

API Types	ROUGE-1	ROUGE-2	ROUGE-SU4
java.util.ArrayList	0.6618	0.4669	0.4855
javax.swing.JFrame	0.6658	0.4033	0.4311
java.lang.NullPointerException	0.6605	0.3761	0.4247
javax.xml.bind.JAXB	0.7296	0.5153	0.5304
java.io.IOException	0.4733	0.2639	0.2848
java.awt.event.ActionListener	0.6587	0.3985	0.4311
java.sql.ResultSet	0.6130	0.3933	0.4128
java.text.SimpleDateFormat	0.5072	0.3302	0.3465
java.math.BigDecimal	0.6188	0.4558	0.4709
java.nio.ByteBuffer	0.6831	0.5512	0.5580
Average	0.6269	0.4152	0.4374

Table 6.6: Results of inter-method comparison, based on ROUGE measures

Method	ROUGE-1	ROUGE-2	ROUGE-SU4
LDA	0.5368	0.2617	0.3051
KM	0.5419	0.2718	0.3142
MMR	0.5724	0.3073	0.3537
LexRank	0.5859	0.31357	0.3569
DISCA	0.6098	0.3431	0.3839

API types, **DISCA** has the best performance for javax.xml.bind.JAXB and the worst performance for java.io.IOException. For javax.xml.bind.JAXB, most candidate API negative caveats discuss issues related to “XML document”, “unmarshal” and “marshal”. These repetitive discussions have more n-gram overlap, which leads to higher ROUGE scores. For java.io.IOException, it is a common IO exception class which can be thrown in many different scenarios. As such, the sentences that discuss IOException have the least level of overlap, which leads to lower ROUGE score.

Inter-Method Comparison. We now compare the relative performance between one method and the other four methods using the Jackknifing procedure [Lin, 2004]. That is, we treat the summary of one method as machine generated summary, and the summaries of the other four methods as reference summaries. Table 6.6 shows the results of inter-method comparison. The table shows that **DISCA** achieves the best performance in all ROUGE scores when the summaries of the other four baseline methods

are used as reference summaries. In contrast, the performance of the other four baseline methods are poorer when the summary of **DISCA** is used as a reference summary. These results indicate that none of the baseline methods can well cover API negative caveats that **DISCA** distills, but **DISCA** covers theirs.

6.4.3 Qualitative Analysis

Comparative Study. In this study, we compare the API negative caveats mentioned in official API documentation and those distilled by **DISCA**. We read the official documentation of the 10 Java API types to annotate the API negative caveats mentioned in these documents. If a distilled API negative caveat and a mentioned API negative caveat both discuss the same aspect of a given API type, we manually judge that they match each other. Table 6.7 reports the results of this comparative study. The three columns show the numbers of negative caveats that are mentioned in official documentation, distilled by **DISCA**, and the matched.

Observe that only 5 out of 10 official documentation mentions about negative caveats and total 6 negative caveats are mentioned. **DISCA** manages to identify 4 out of the 6 mentioned negative caveats. One missed negative caveat is about rounding behavior of `java.math.BigDecimal` class.⁶ The other missed negative caveat is from `javax.swing.JFrame`.⁷ By checking our dataset, we find that none of the candidate API negative caveats for `BigDecimal` mention “rounding mode” or relevant concepts; the same observation holds for `JFrame`. Searching Stack Overflow website using queries “`BigDecimal` rounding mode” and “`JFrame` serialized objects” results in 105 and 45 posts, respectively. We do not find any negative sentences discussing the two issues in the search results. The results suggest that the two missed API negative caveats have not been well discussed on Stack Overflow.

For the 10 API types, **DISCA** distills 164 negative caveats related to 46 semantic aspects of the 10 API types. Recall that each term-community is considered as a semantic aspect of an API, and it may have several complementary API negative caveats

⁶If no rounding mode is specified and the exact result cannot be represented, an exception is thrown. <https://docs.oracle.com/javase/8/docs/api/java/math/BigDecimal.html>

⁷Serialized objects of this class will not be compatible with future Swing releases. <https://docs.oracle.com/javase/8/docs/api/javax/swing/JFrame.html>

Table 6.7: The three columns show the numbers of negative caveats that are mentioned in official documentation, distilled by **DISCA**, and the matched between the two

API Types	Mentioned	Matched	DISCA
java.util.ArrayList	1	1	18
javax.swing.JFrame	1	0	17
java.lang.NullPointerException	0	0	24
javax.xml.bind.JAXB	0	-	16
java.io.IOException	0	-	18
java.awt.event.ActionListener	0	-	17
java.sql.ResultSet	1	1	14
java.text.SimpleDateFormat	2	2	22
java.math.BigDecimal	1	0	12
java.nio.ByteBuffer	0	-	6
Sum	6	4	164

(see Figure 6.5). We manually examine the 164 negative caveats and find that there are only 18 false positive instances. There are two main reasons for false positive instances. First, **DISCA** distills 7 programming exceptions as API negative caveats. For example, one of the false positive instances is “*Exception: IOException is not compatible with throws clause in Plants.eat()*”. This is an exception related to the implementation of a specific program `Plants.eat()`, but not the API `IOException`. Second, 11 false positive instances are context-dependent sentences that our sentence filtering patterns fail to filter out. For example, in the sentence “*you cannot attach an ActionListener without having to rewrite the controller and the view*”, “the controller” and “the view” refer to other parts of the discussion. As such, this sentence is hard to understand without additional context. Although there is a small percentage (about 11%) of false positive instances, **DISCA** distills 146 correct API negative caveats which can drastically augment the 10 official API documents.

Case Study. As a case study, we present the distilled API negative caveats of four Java API types. We select three API types in Table 6.2: `javax.swing.JFrame`, `java.awt.event.ActionListener`, and `java.math.BigDecimal`, which are mentioned frequently, moderately, and relatively infrequently. We also include `java.util.HashMap` which has been used to illustrate our approach throughout our discussion. For the four API types, **DISCA** detects 4, 6, 5 and 3 term-communities, respectively. For each term-community, we list the

Table 6.8: Examples of API negative caveats mined by DISCA

java.util.HashMap
(1.1) HashMap does not provide any guarantees of order among its entries.
(1.2) HashMap is not thread safe for concurrent access.
(1.3) HashMap does not need keys to be Comparable but still implements Map interface.
(1.4) HashMap cannot store multiple values for the same key.
javax.swing.JFrame
(2.1) Don't extend a JFrame, but instead create a local JFrame variable and use it.
(2.2) JFrame isn't focusable JComponent, you would need to use focusable container <i>e.g.</i> , JPanel.
(2.3) JFrame does not extend JComponent and does not have a paintComponent method.
(2.4) You shouldn't set a JFrame visible until all the components have been added.
(2.5) Calling validate on a top-level component (JWindow, JDialog, JFrame) will not necessarily resize that component.
(2.6) Don't call JFrame#setSize(..) on JFrame rather just call JFrame#pack() before setting JFrame visible.
java.awt.event.ActionListener
(3.1) You can't add an ActionListener to a JFrame, it does not function like a button and so has no action listeners.
(3.2) An ActionListener can't be added to a JPanel, as a JPanel itself does not lend itself to create what is considered to be "actions".
(3.3) Don't implement ActionListener in top classes, use anonymous classes or private classes instead.
(3.4) Don't implement single ActionListener for multiple components.
(3.5) An ActionListener cannot distinguish states on it's own, it simply responds to a user input.
java.math.BigDecimal
(4.1) BigDecimal is not a primitive type.
(4.2) BigDecimal cannot support numbers that cannot be written as a fixed length decimal, <i>e.g.</i> , 1/3.
(4.3) Unlike Integer and Double, BigDecimal does not participate in autoboxing.

top-1 ranked API negative caveat as a representative caveat in Table 6.8.

For java.util.HashMap, its long official documentation mentions three negative caveats related to *element order*, *multiple threads synchronization* and *comparable element*. Only the sentence for *multiple threads* is in bold text. Compared with the lengthy official documentation, the API negative caveats (1.1), (1.2) and (1.3) in Table 6.8 show that DISCA helps to surface hard-to-notice API negative caveats. Furthermore, DISCA augments the official documentation with caveat (1.4) about *multiple values for the same key*. This may seem natural, but often overlooked. Making it explicit provides an im-

portant reminder for novice developers.

For `javax.swing.JFrame`, all the mined API negative caveats by **DISCA** do not exist in its Javadoc. Caveats (2.1) and (2.2) caution users about not extending `JFrame` and `JFrame` being not focusable. Moreover, they give alternative solutions at the same time. Caveat (2.3) emphasizes that `JFrame` has no `paintComponent` method. Caveats (2.4), (2.5) and (2.6) are on the issues related to setting `JFrame` visible, validating `JFrame`, and setting `JFrame` size, respectively. These API negative caveats are difficult to be documented by API designers, because they mainly emerge from misuse in practice.

For `java.awt.event.ActionListener`, its Javadoc does not mention any API negative caveats. **DISCA** distills 5 negative caveats. Caveats (3.1) and (3.2) not only caution users that they cannot add `ActionListener` to `JFrame` or `JPanel` but also explain the reason behind. Caveats (3.2), (3.3) and (3.5) are good but implicit coding practices when implementing `ActionListener`. It is infeasible for API designers to take all these aspects into account when documenting API, because these API negative caveats can only be accumulated in practice.

For `java.math.BigDecimal`, although **DISCA** does not find API negative caveat regarding *round mode* of `BigDecimal`, it finds three other negative caveats. Caveat (4.1) warns developers that “*BigDecimal is not a primitive type*”. Similarly, caveats (4.2) and (4.3) provide two important usage cautions about “*fixed length decimal*” and “*autoboxing*”. These to-be-avoided usage contexts are hard to foresee.

6.5 Tool Implementation

To facilitate exploration of the discovered API caveats, we implemented a web interface⁸ with three interactive modes, using `d3js`⁹ toolkit and `FoamTree`¹⁰ framework. The main GUI consists of five components, as indicated on Figure 6.7.

- (i) *The query input panel* is implemented with an API-aware auto-complete method, and the matched queries are displayed as user-friendly responsive drop-down menu.

⁸<http://138.197.118.157:8000/caveat/>

⁹<https://d3js.org/>

¹⁰<https://carrotsearch.com/foamtree/>

Figure 6.7: **DISCA** screenshots (best viewed in color). The main GUI consists of five components: ① The query input panel. ② Interactive modes: *FoamTree*, *Circles*, and *Fisheye*. ③ Term community, representing a semantic aspect of an API. ④ Link to the original post on Stack Overflow. ⑤ The sentence detailing the API caveat.

- (ii) *The interactive panel* provides three modes to explore the discovered API caveats: *FoamTree* mode, *Circles* mode and *Fisheye* model, to be detailed shortly.
- (iii) *The term community* shows all prominent terms in a cluster. Each community represents one key semantic aspect of the API type.
- (iv) *The link* to the original post on Stack Overflow, from which this caveat is extracted, for accessing the discussion thread.
- (v) *The caveat* shows the sentence (*i.e.*, API caveat) discovered by **DISCA**, which mentions the queried API and contains negative expressions.

DISCA implements three interactive modes. *FoamTree* mode provides an engaging user experience with animated transitions and zooming, as an effective way to explore the details of API caveats. Unlike *FoamTree* mode which needs zooming, *Circles* mode provides the whole picture of discovered API caveats within a highly-interactive multi-level pie chart. *Fisheye* mode uses an interactive graph to show the relationships among the terms of API caveats. The node size is proportional to the

degree centrality of the node in the graph. Different colors indicate different term communities. In short, the three interactive modes together facilitate users to explore and understand the discovered API caveats.

6.6 Threats to Validity

This section outlines potential threats to the validity of this study.

The Use of Data. All APIs investigated in our evaluation are Java JDK APIs, and our evaluation uses only Stack Overflow discussions. The framework of **DISCA** makes no specific assumptions about APIs and discussion data. Therefore, it is generally applicable for other programming languages or third-party libraries or other Q&A websites, which we leave as our future work. Our quantitative evaluation is based on gold standard summaries generated by human annotators. They could be biased. But our inter-annotator agreement analysis suggests that the gold standard summaries used in the evaluation are acceptable.

The Coverage of Candidate Sentences. First, **DISCA** currently uses a simple name matching strategy to select candidate sentences. Entity linking approaches [Ji et al., 2016, Ye et al., 2016b, Moro et al., 2014] based on machine learning could improve the performance of **DISCA**, because these approaches can better handle API-mention variations and thus provide more candidate sentences for selection.

Second, based on our observation, important information about a caveat mostly appears in a single sentence on Stack Overflow. **DISCA** may miss some multiple-sentences caveats because **DISCA** is designed at single sentence level. However, it is infeasible to get the number of multiple-sentence caveats without manual annotation of the data. We show one example here: *“Iterator returned by HashMap are fail-fast while Enumeration returned by the HashTable are fail-safe. Fail-safe is relevant from the context of iterators. If an iterator has been created on a collection object and some other thread tries to modify the collection object “structurall”, a concurrent modification exception will be thrown.”* These three sentences are about the caveat of *“thread safe”*. Although our approach will not extract these three sentences, it will extract *“HashMap is not thread safe for concurrent access.”* from many other single-sentence candidates for this caveat.

Third, the sentence selection process of **DISCA** may result in missing some caveats because of some strict restrictions (*e.g.*, removing subjective opinions in Section 6.3.1, selecting explicitly negative sentences in Section 6.3.2, selecting prominent terms in Section 6.3.4). Given the thousands of sentences to be examined during the sentence selection process, annotating intermediate results and analyzing influence factors requires much human efforts, if even feasible. As many sentences are available for important caveats, our proposed approach is able to pick up the representative sentences even though some sentences are missed.

6.7 Related Work

Given the commonality of crowd-generated content, our related work section is divided into three parts: works on crowdsourced knowledge in software development, App review opinion mining, and automatic text summarization.

6.7.1 Crowdsourced knowledge in software development

Several studies have contributed effort on aiding developers in software development using crowdsourced knowledge. The crowdsourced knowledge generally derived from two types of sources: *code snippet* and *textual content*. For crowdsourced code snippets, there are studies [Sahavechaphan and Claypool, 2006, Thummalapenta and Xie, 2007] investigating the problem that how to integrate code snippets in Integrated Development Environment (IDE) to recommend code examples during software development. For example, XSnippet [Sahavechaphan and Claypool, 2006], a context-sensitive code assistant framework, allows developers to query a sample repository for code snippets that are relevant to the programming task on hand. Other studies [Brandt et al., 2010, Kim et al., 2009, Zagalsky et al., 2012] investigated the problem that how to build up a new code search by utilizing source snippets on the Web. In addition, a few studies focused on recovering the traceability of various software artefacts, such as the link between source code snippets and official API documentation [Kim et al., 2009, Subramanian et al., 2014] and the link between source code snippets and their learning resources [Dagenais and Robillard, 2012]. Such existing

studies on crowdsourced code snippet are based on code search engines and code static analysis. In contrast, **DISCA** is designed to distill negative caveats which are expressed in natural language.

Another type of crowdsourced knowledge is textual content. Many studies developed tools to integrate Q&A resources into the IDE, such as Seahawk [Bacchelli et al., 2012], and Prompter [Ponzanelli et al., 2013a, Ponzanelli et al., 2014a]. Studies also developed question answering systems to answer programming questions by leveraging official content and social context of software documentation [Li et al., 2016, Li et al., 2018a, Li et al., 2018b]. In addition, researchers have contributed their efforts for program comprehension by using software textual content [Ponzanelli et al., 2013a, Treude et al., 2011]. These existing studies link or recommend crowdsourced knowledge from the point view of “how to use an API” at post level or document level. In contrast, our work distills insights at fine-grained sentence level from the point view of “how not to use an API”. Recently, Treude *et al.* [Treude and Robillard, 2016] presented SISE, which automatically augments API documentation with insight sentences from Stack Overflow. This work is the closest to our work. However, in [Treude and Robillard, 2016], the authors trained a binary classifier with hand-coded features in a supervised manner and the solution does not consider the factors of redundancy, diversity, and negative expression in summarization algorithm. In short, existing studies focus on general relevance of the recommended knowledge, while our work specifically focuses on negative insights related to API usages. To the best of our knowledge, no prior work has been done on negative usages of APIs.

6.7.2 App review opinion mining

Our work is to summarize the crowd-generated sentences with respect to APIs. It is similar to crowd-generated reviews for Apps. Miao *et al.* [Miao et al., 2010] exploited the domain knowledge to assist product feature extraction and sentiment orientation identification from unstructured reviews. Wisniewski *et al.* [Wisniewski et al., 2015] examined two prominent Facebook features that promote confidant disclosures: tagging and third-party applications. The results illustrate the complexity of the trade-off

between privacy concerns, engaging with friends through tagging and Apps, and Facebook usage. Gu *et al.* [Gu and Kim, 2015] presented SURMiner, which classifies reviews into five categories (*i.e.*, aspect evaluation, bug reports, feature requests, praise and others) and extracts aspects in sentences using a pattern-based parser. Chen *et al.* [Chen et al., 2014] developed AR-Miner, which helps App developers extract the most valuable information from raw user review data. Vu *et al.* [Vu et al., 2015] proposed MARK, a keyword-based framework for semi-automated review summarization and visualization.

These existing work focuses on opinion-aspect phrase extraction [Vu et al., 2016] and conducts sentiment analysis of opinion words [Gu and Kim, 2015, Serva et al., 2015]. Although API negative caveats are expressed in negative sentences, they state a neutral fact about API usage rather than a polarity opinion. Thus, sentiment analysis followed by aspect extraction in the above work is generally not applicable for distilling API negative caveats from crowd-generated discussions.

6.7.3 Automatic text summarization

In recent years, there has been an explosion in the amount of text data, which needs to be effectively summarized to be useful. Those existing approaches in general fall into two categories: *extractive summarization* and *abstractive summarization*. Extractive summarization methods select a few relevant sentences from the original document as a summary. Summary sentence selection therefore is a critical step in the extractive summarization process. Most previous shallow models estimate the salience of a sentence using predefined features, such as lexical chains [Barzilay and Elhadad, 1999], word co-occurrence [Matsuo and Ishizuka, 2004] and centrality [Erkan and Radev, 2004]. Recently, many advanced models are developed to learn deep semantic features. For example, Cao *et al.* [Cao et al., 2015] developed PriorSum, which applies enhanced convolutional neural networks to capture the summary prior features derived from length-variable phrases. The learned prior features are concatenated with document-dependent features for sentence ranking. Ren *et al.* [Ren et al., 2017] proposed a neural extractive model, named contextual relation-based summarization, to take advantage

of contextual relations among sentences so as to improve the performance of sentence regression.

Abstractive summarization methods produce a new concise text which includes words and phrases different from the ones in the source document. Structure-based approaches have been studied extensively, such as rule-based [Genest and Lapalme, 2012], ontology-based [Lee et al., 2005], and template-based [Harabagiu and Laca-tusu, 2002] approaches. Recently, semantic-based approaches are widely investigated. Bing *et al.* [Bing et al., 2015] proposed an abstractive multi-document summarization framework that can construct new sentences by exploring more fine-grained syntactic units than sentences. Nallapati *et al.* [Nallapati et al., 2016] proposed an abstractive text summarization model using attentional encoder-decoder recurrent neural networks. Paulus *et al.* [Paulus et al., 2017] proposed a neural network model with a novel intra-attention that attends over the input and continuously generated output separately. This model combines standard supervised word prediction and reinforcement learning for abstractive summarization. Tan *et al.* [Tan et al., 2017] proposed a graph-based attention mechanism in the sequence-to-sequence framework. This framework introduced a new hierarchical decoding algorithm with a reference mechanism to generate the abstractive summaries.

Although these existing studies leverage advanced NLP techniques to generate summaries, they require a great amount of training data. For the research problem in this paper, there is no training data available. The advantage of our framework is that it is an unsupervised and a data-driven method.

6.8 Summary

This research identifies a new task of distilling crowdsourced API negative caveats from a large volume of programming-related discussion. We present an effective text-summarization approach to distilling context-independent, prominent, semantically diverse and non-redundant API negative caveats. Our approach significantly outperforms other text-summarization methods, including the methods that are based on eigenvector centrality of sentence graph, topic modeling, sentence clustering, and

sentence diversification. Furthermore, our approach greatly augments official API documentation with crowdsourced API negative caveats and explanations, as well as suggestions (*e.g.*, alternative APIs) for solving API usage issues. We are developing web applications that can push distilled API negative caveats when developers read API documents. All APIs investigated in our evaluation are JDK APIs, and our evaluation uses only Stack Overflow discussions. However, we argue that the framework of **DISCA** makes no specific assumptions on the input data, *e.g.*, APIs and discussion. It is generic to other programming languages or other Q&A websites. **DISCA** currently uses a simple name matching strategy to select candidate sentences. Entity linking approaches [Ji et al., 2016, Ye et al., 2016b, Moro et al., 2014] based on machine learning could improve the performance of **DISCA**, because these approaches better handle API-mention variations which will lead to more candidate sentences for selection. As a part of future study, we will mine programming errors related to API negative caveats to develop semantic search systems that can provide direct answers to such errors caused by overlooking API negative caveats.

Conclusion and Future Work

7

Prediction is very difficult, especially if it's about the future.

— Biels Bohr

All dots are connected.

— Steve Jobs

In this chapter, we first summarize the work this thesis completed and then we outline some of research directions stemming from this thesis.

7.1 Conclusion

In this thesis, we have taken several steps to understand information needs of developers. We also formulated three tasks to address the problem on supporting information needs of developers.

First, we conducted an empirical study to understand information needs of developers based on 29 hours of screen-captured task videos of 20 developers. This work reveals three key insights on the information gap when developers seek and use web resources: (1) developers might have an incomplete or even incorrect understanding of their needs; (2) there is a gap between the producers and consumers of software documentation; (3) many important pieces of information that developers need are explicitly undocumented in software documentation. These findings inspire us to conduct the following work.

Second, we developed LinkLive technique to recommend more correlated learning resources when developers know less. LinkLive uses multiple features, including hyperlink co-occurrences in web Q&A discussions, locations (*e.g.*, question, answer, or

comment) in which hyperlinks are referenced, and votes for posts/comments in which hyperlinks are referenced. A large-scale evaluation shows that our technique recommends correlated web resources with satisfactory precision and recall in an open setting.

Third, we proposed a novel *deep-learning-to-answer* framework, named QDLinker, for answering programming questions with software documentation. QDLinker leverages the large volume of discussions in Community-based Question Answering to bridge the semantic gap between programmers' questions and software documentation. Through extensive experiments, we show that QDLinker significantly outperforms the baselines based on traditional retrieval models and Web search services dedicated for software documentation.

Fourth, we proposed DISCA, a novel approach to automatically distilling desirable API negative caveats from unstructured web Q&A discussions. The quantitative and qualitative evaluations show that DISCA can greatly augment the official API documentation.

7.2 Future Work

In this thesis, we have an understanding of information needs of developers towards Web use at the micro-level. We also conducted three studies to develop approaches to support developers' information needs. However, there are still many issues we did not address. In future, we are going to investigate the following directions:

Comparative APIs mining. On Stack Overflow, developers always discuss features of different APIs. These discussions include some comparative sentences about API feature. For example, *"Don't use a HashMap if you are going to have multiple threads, use a ConcurrentHashMap instead."* and *"Unlike Integer and Double, BigDecimal does not participate in autoboxing."* The crowdsourced data makes it possible to mine comparative relation of APIs. The relation can help developers quickly master unfamiliar APIs, because developers can distinguish the similarities and differences of two APIs with these comparative relations. Meanwhile, these comparative relations can be augmented in official documentation to call developers' attentions.

Automatically finding fixes for API-usage-bug caused by API caveats. In Chapter 6, we have mined API negative caveats from web Q&A discussions. We also observe that some developers posted different programming questions which were caused by some API caveats. We call these questions as “API-usage-bugs”. Obviously, these API-usage-bugs are independent of the functional requirements. As shown in Chapter 6, API-usage-bugs are very likely to result in unexpected programming errors because of an incorrect usage of an API. These API-usage-bugs occur repeatedly in different contexts and they are independent of the application domain. This direction aims to automatically find fixes for these API-usage-bug caused by API caveats.

Deep API documentation reader. In Chapter 5, we have developed a model to automatically answer programming questions with software documentation. However, it is a coarse-grained approach at document level. Sometimes, the API documentation is very verbose so that developers must scan it carefully to identify the answer. The direction aims to provide a *direct answer* from the documentation through a machine reader. More specifically, given a document, a question and an answer, we want to train a reader which can capture the relations among them. For example, for question “*what is the default layout in Dialog?*” and a document “*....The size of the frame includes any area designated for the border. The default layout for a frame is BorderLayout...*”, the ultimate goal is to learn the answer *BorderLayout* from the document. Essentially, it is a reading comprehension task for API documentation.

A

Appendix

A.1 Evaluation results of scvRipper

In Chapter 3, we used video scraping tool (*scvRipper*) to collect data from screen-captured task videos in our empirical studies. In the appendix, we summarize the effectiveness and runtime performance of our *scvRipper*.

Figure A.1: The screenshot of *scvRipper*

Figure A.1 shows the Graphical User Interface (GUI) of the *scvRipper* tool. The analyst can select a screen-captured video. The *scvRipper* tool parses the video and detect distinct-content screenshots. It lists the distinct-content screenshots in the left panel. The analyst can analyze one screenshot at a time or analyze all the screenshot in batch mode. The *scvRipper* tool visualizes the intermediate image processing results in the right panel, such as the detected horizontal and vertical lines, the detected visual

cues, and the detected window boundaries. The analyst can zoom-in and inspect these intermediate results to determine the quality of the video-scraped data.

Runtime Performance. We ran our *scvRipper* tool on a Windows 7 computer with 4GB RAM and Intel(R) Core(TM)2 Duo CPU. The 29 hours task videos were recorded at sample rate 5 screenshots per second. As such, the 29 hours task videos consists of in total over 520K screenshots. Our *scvRipper* tool took 43 hours to identify about 11K distinct-content screenshots from the 29 hours videos at the threshold $t_{diff} = 0.7$. One distinct-content screenshot on average represents about 10 seconds video. The *scvRipper* tool took about 122 hours to extract time-series interaction data from the 11K distinct-content screenshots, i.e., on average 38.41 ± 16.94 seconds to analyze one distinct-content screenshot. The OCR of the scraped image content took about 60 hours .

The current implementation of the *scvRipper*'s core algorithm processes one distinct-content screenshot at a time (i.e., sequential processing). The most time-consuming step of the core algorithm is the second step (i.e., detect individual visual cues). Our definition of the Eclipse IDE and Chrome window consists of about 30 and 20 visual cues respectively. The current implementation detects visual cues in a screenshot one at a time. This step consumes about 97% of the processing time of distinct-content screenshots. Since the processing of individual screenshots and the detection of individual visual cues are independent, the runtime performance of the *scvRipper* tool could be significantly improved by parallel computing [Quinn, 1994] and hardware-implementation of template-matching algorithm [Sinha et al., 2006]. Parallel computing and hardware acceleration ¹ could also reduce the time of detecting distinct-content screenshots and the OCR of scraped screen images.

Effectiveness. We randomly sampled 500 distinct-content screenshots from different developers' task videos at different time periods. We qualitatively examined the screenshots that these sampled distinct-content screenshots represent. We found that the *scvRipper*'s image differencing technique (at $t_{diff} = 0.7$ in this study) can tolerate

¹<http://docs.opencv.org/modules/gpu/doc/introduction.html>

the reasonable differences between the screenshots caused by scrolling, mouse movement, and pop-up menus. Ignoring these screenshots should not cause significant information loss for data analysis.

We qualitatively examined the results of detected application windows in these sampled distinct-content screenshots. Our *scvRipper* tool sometimes may miss certain visual cues. As long as some visual cues were detected (over 80% of defined VisualCues in this study), *scvRipper* usually can still recognize the application window. However, missing some visual cues may result in the less accurate detection of window boundary. For example, the detected window boundary may miss the title bar due to the failure of detecting the corresponding title bar visual cue. Our *scvRipper* tool can accurately recognize side-by-side or stacked windows. But it cannot accurately detect several (≥ 3) overlapping windows, each of which is only partially visible. However, screenshots with several overlapping windows are rare in our dataset.

We evaluated the accuracy of the OCR results using the extracted query keywords. *scvRipper* identified 236 distinct-content screenshots that contain a search query. These queries contain 253 English words and 809 Chinese words in total. The OCR accuracy of the English words is about 88.5% (224/253), while the OCR accuracy of the Chinese words is about 74.9% (606/809). The screenshots had low DPI (Dots Per Inch, only 72-96 DPI in participants' computer) which is lower than the 300 DPI that the OCR tool generally requires. The OCR tool (ABBYY FineReader) we used scaled the low DPI screenshots to 300 DPI and produced acceptable OCR results.

A.2 Predefined patterns for filtering out context-dependent sentences

- Sentences in question body

Example: *ListView adapter with HashMap isn't displaying correctly.*

- Interrogative sentences

Example: *Have you overridden the `keySet()` method in your HashMap?*

- References to code: “: code”, “the above cod”, etc.

Example: *An equivalently synchronised HashMap can be obtained by: code.*

- References about "do this", "do that", "like this", "like that""for this", "for that",etc.
Example: *If you are trying to do this in a single thread, I would recommend HashMap.*
You cannot delete from HashMap like that.
Also, may be HashMap will be more useful for that task.
- References to another sentence: "see the next step", "the following",etc.
Example: *See the next step if you need a mutable list.*
The following article addresses this question in some detail: HashMap requires a better hashCode().
- References about "in this way", "in that way", "in this case", "in that case",etc.
Example: *Note that i am creating multiple jdbcTemplates in this way and maintaining them in a HashMap.*
The HashMap on Class won't work in that case because the Class won't be found.
- References starts with "This+verb", "That+verb",etc.
Example: *That depends on what you want to do if the value is not contained in the HashMap.*
- References about "as you're doing", "what you are doing",etc.
Example: *Then, in the while loop, get the current inner ArrayList as you're doing.*
- References about Stack Overflow users :"@user", another docments: "@document",etc.
Example: *If it's not annotated with @XmlElement, then JAXB does not have sufficient information to marshal it.*
- Sentences have no subjects
Example: *Can't add to HashMap.*
- Subjective opinion: "I'm not sure", "I don't think", "I don't understand", "I don't know", "in my opinion", "I think",etc.
Example: *I don't think you should extend HashMap, you should manage an existing Map implementation from the outside.*
I'm not sure about HashMap implementation but I think it uses open addressing too.

Publications

All publications of doctoral period are listed as follows. Note that the publications of 1~7 are included in this thesis.

- (1) Leveraging Official Content and Social Context to Recommend Software Documentation

Jing Li, Zhenchang Xing and Ashad Kabir. *IEEE Transactions On Services Computing*. DOI: 10.1109/TSC.2018.2812729. (IF 3.520).

- (2) Learning to Answer Programming Questions with Software Documentation through Social Context Embedding

Jing Li, Aixin Sun and Zhenchang Xing. *Information Science*. Volumes 448-449, Pages 36-52, June 2018, Elsevier. (IF 4.832)

- (3) LinkLive: Discovering web learning resources for developers from Q&A discussions

Jing Li, Zhenchang Xing and Aixin Sun. *World Wide Web Journal*. Springer. Minor Revision.

- (4) To Do or Not To Do: Distill Crowdsourced Negative Caveats to Augment API Documentation

Jing Li, Aixin Sun and Zhenchang Xing. *Journal of the Association for Information Science and Technology*. Wiley, 2018. Accepted.

- (5) API Caveat Explorer: Surfacing Negative Usages from Practice

Jing Li, Aixin Sun, Zhenchang Xing and Lei Han.
SIGIR 2018 Demo. Accepted.

- (6) From Discussion to Wisdom: Web Resource Recommendation for Hyperlinks in Stack Overflow

Jing Li, Zhenchang Xing, Deheng Ye and Xuejiao Zhao. *The 31st ACM Symposium on Applied Computing (SAC 2016)*. pp. 1127-1133. AR: 252/1047 (24.07%).

- (7) BPMiner: Mining Developers' Behavior Patterns from Screen-Captured Task Videos

Jing Li, Lingfeng Bao, Zhenchang Xing, Xinyu Wang and Bo Zhou. *The 31st ACM Symposium on Applied Computing (SAC 2016)*. pp. 1371-1377. AR: 252/1047 (24.07%).

- (8) SegBot: A Generic Neural Text Segmentation Model with Pointer Network

Jing Li, Aixin Sun and Shafiq Joty. IJCAI 2018. Accepted. AR: 710/3470 (20.5%).

- (9) Extracting and Analyzing Time-Series HCI Data from Screen-Captured Task Videos

Lingfeng Bao, **Jing Li**, Zhenchang Xing, Xinyu Wang, Xin Xia and Bo Zhou. *Empirical Software Engineering (EMSE Journal)*, Springer, 2016. pp.1-41.

- (10) scvRipper: Video Scraping Tool for Modeling Developers' Behavior Using Interaction Data

Lingfeng Bao, **Jing Li**, Zhenchang Xing, Xinyu Wang and Bo Zhou. *The 37th International Conference on Software Engineering (ICSE 2015) Tool Demo*, vol.2, pp. 673-676.

- (11) Reverse Engineering Time-Series Interaction Data from Screen-Captured Videos

Lingfeng Bao, **Jing Li**, Zhenchang Xing, Xinyu Wang and Bo Zhou. *The 22nd IEEE International Conference on Software Analysis, Evolution, and Reengineering (SANER 2015)*, pp. 399-408. AR: 46/144 (32%).

- (12) Learning to Extract API Mentions from Informal Natural Language Discussions

Deheng Ye, Zhenchang Xing, Chee Yong Foo, **Jing Li** and Nachiket Kapre. *The 32nd International Conference on Software Maintenance and Evolution (ICSME 2016)*. Vol. 1, pp. 90-101. AR: 37/125 (29%).

- (13) Software-specific Named Entity Recognition in Software Engineering Social Content
Deheng Ye, Zhenchang Xing, Chee Yong Foo, Zi Qun Ang, **Jing Li** and Nachiket Kapre. *The 23rd IEEE International Conference on Software Analysis, Evolution, and Reengineering (SANER 2016)*. Vol. 1, pp. 90-101. AR: 52/140 (37%).
- (14) Software-specific Part-of-speech Tagging: An Experimental Study on Stack Overflow
Deheng Ye, Zhenchang Xing, **Jing Li** and Nachiket Kapre. *The 31st ACM Symposium on Applied Computing (SAC 2016)*. pp. 1378-1385. AR: 252/1047 (24.07%).
- (15) HDSKG: Harvesting Domain Specific Knowledge Graph from Content of Webpages
Xuejiao Zhao, Zhenchang Xing, Muhammad Ashad Kabir, Shangwei Lin, **Jing Li** and Naoya Sawada. *The 24th IEEE International Conference on Software Analysis, Evolution, and Reengineering (SANER 2017)*. AR: 34/140 (24.3%).

References

- [Aho and Hopcroft, 1974] Aho, A. V. and Hopcroft, J. E. (1974). *The design and analysis of computer algorithms*. Pearson Education India.
- [Anderson et al., 2012] Anderson, A., Huttenlocher, D., Kleinberg, J., and Leskovec, J. (2012). Discovering value from community activity on focused question answering sites: a case study of stack overflow. In *Proc. of KDD*, pages 850–858.
- [Bacchelli et al., 2011] Bacchelli, A., Cleve, A., Lanza, M., and Mocci, A. (2011). Extracting structured data from natural language documents with island parsing. In *Proc. of ASE*, pages 476–479.
- [Bacchelli et al., 2012] Bacchelli, A., Ponzanelli, L., and Lanza, M. (2012). Harnessing stack overflow for the ide. In *Proc. of the Third International Workshop on Recommendation Systems for Software Engineering*, pages 26–30.
- [Baeza-Yates et al., 2015] Baeza-Yates, R., Boldi, P., and Chierichetti, F. (2015). Essential web pages are easy to find. In *Proc. of WWW*, pages 97–107.
- [Bagheri and Ensan, 2014] Bagheri, E. and Ensan, F. (2014). Semantic tagging and linking of software engineering social content. *Automated Software Engineering*, pages 1–44.
- [Bajracharya et al., 2010] Bajracharya, S. K., Ossher, J., and Lopes, C. V. (2010). Leveraging usage similarity for effective retrieval of examples in code repositories. In *Proc. FSE*, pages 157–166.
- [Barua et al., 2014] Barua, A., Thomas, S. W., and Hassan, A. E. (2014). What are developers talking about? an analysis of topics and trends in stack overflow. *Empirical Software Engineering*, 19(3):619–654.

- [Barzilay and Elhadad, 1999] Barzilay, R. and Elhadad, M. (1999). Using lexical chains for text summarization. *Advances in automatic text summarization*, pages 111–121.
- [Begel and Zimmermann, 2013] Begel, A. and Zimmermann, T. (2013). Analyze this! 145 questions for data scientists in software engineering. In *Microsoft Research. Technical Report. MSR-TR-2013-111*, Microsoft.
- [Begel and Zimmermann, 2014] Begel, A. and Zimmermann, T. (2014). Analyze this! 145 questions for data scientists in software engineering. In *Proc. of ICSE*, pages 12–23. ACM.
- [Berger et al., 2000] Berger, A., Caruana, R., Cohn, D., Freitag, D., and Mittal, V. (2000). Bridging the lexical chasm: statistical approaches to answer-finding. In *SIGIR*, pages 192–199.
- [Bernstein et al., 2012] Bernstein, M. S., Teevan, J., Dumais, S., Liebling, D., and Horvitz, E. (2012). Direct answers for search queries in the long tail. In *Proc. of SIGCHI*, pages 237–246.
- [Beyer and Pinzger, 2016] Beyer, S. and Pinzger, M. (2016). Grouping android tag synonyms on stack overflow. In *Proc. of MSR*, pages 430–440.
- [Bing et al., 2015] Bing, L., Li, P., Liao, Y., Lam, W., Guo, W., and Passonneau, R. J. (2015). Abstractive multi-document summarization via phrase selection and merging. In *Proc. of ACL*, pages 1587–1597.
- [Blanco et al., 2013] Blanco, R., Cambazoglu, B. B., Mika, P., and Torzec, N. (2013). Entity recommendations in web search. In *Proc. of ISWC*, pages 33–48.
- [Blei et al., 2003] Blei, D. M., Ng, A. Y., and Jordan, M. I. (2003). Latent dirichlet allocation. *Journal of machine Learning research*, 3(Jan):993–1022.
- [Blondel et al., 2008] Blondel, V. D., Guillaume, J.-L., Lambiotte, R., and Lefebvre, E. (2008). Fast unfolding of communities in large networks. *Journal of statistical mechanics: theory and experiment*, 2008(10):P10008.

- [Brandt et al., 2010] Brandt, J., Dontcheva, M., Weskamp, M., and Klemmer, S. R. (2010). Example-centric programming: integrating web search into the development environment. In *Proc. CHI*, pages 513–522.
- [Brandt et al., 2009] Brandt, J., Guo, P. J., Lewenstein, J., Dontcheva, M., and Klemmer, S. R. (2009). Two studies of opportunistic programming: interleaving web foraging, learning, and writing code. In *Proc. of CHI*, pages 1589–1598.
- [Bretherton, 1985] Bretherton, I. (1985). Attachment theory: Retrospect and prospect. *Monographs of the society for research in child development*, pages 3–35.
- [Broder, 2002] Broder, A. (2002). A taxonomy of web search. In *ACM Sigir forum*, volume 36, pages 3–10.
- [Burke et al., 1997] Burke, R. D., Hammond, K. J., Kulyukin, V., Lytinen, S. L., Tomuro, N., and Schoenberg, S. (1997). Question answering from frequently asked question files: Experiences with the faq finder system. *AI magazine*, 18(2):57.
- [Buse and Zimmermann, 2012] Buse, R. P. and Zimmermann, T. (2012). Information needs for software development analytics. In *Proc. ICSE*, pages 987–996.
- [Cao et al., 2010] Cao, X., Cong, G., Cui, B., and Jensen, C. S. (2010). A generalized framework of exploring category information for question retrieval in community question answer archives. In *WWW*, pages 201–210.
- [Cao et al., 2015] Cao, Z., Wei, F., Li, S., Li, W., Zhou, M., and Wang, H. (2015). Learning summary prior representation for extractive summarization. In *Proc. of ACL*, pages 829–833.
- [Chen et al., 2017] Chen, C., Xing, Z., and Wang, X. (2017). Unsupervised software-specific morphological forms inference from informal discussions. In *Proc. of ICSE*.
- [Chen et al., 2014] Chen, N., Lin, J., Hoi, S. C., Xiao, X., and Zhang, B. (2014). Arminer: mining informative reviews for developers from mobile app marketplace. In *Proc. of ICSE*, pages 767–778.

- [Cheng et al., 2015] Cheng, Y., Yu, F. X., Feris, R. S., Kumar, S., Choudhary, A., and Chang, S.-F. (2015). An exploration of parameter redundancy in deep networks with circulant projections. In *Proc. of ICCV*, pages 2857–2865.
- [Collobert et al., 2011] Collobert, R., Weston, J., Bottou, L., Karlen, M., Kavukcuoglu, K., and Kuksa, P. (2011). Natural language processing (almost) from scratch. *Journal of Machine Learning Research*, 12(Aug):2493–2537.
- [Cooley et al., 1997] Cooley, R., Mobasher, B., and Srivastava, J. (1997). Web mining: Information and pattern discovery on the world wide web. In *Proc. of ICTAI*, pages 558–567.
- [Cordeiro et al., 2012] Cordeiro, J., Antunes, B., and Gomes, P. (2012). Context-based recommendation to support problem solving in software development. In *Proc. of Recommendation Systems for Software Engineering*, pages 85–89.
- [Dagenais and Robillard, 2012] Dagenais, B. and Robillard, M. P. (2012). Recovering traceability links between an api and its learning resources. In *Proc. of ICSE*, pages 47–57.
- [DeLine et al., 2005] DeLine, R., Czerwinski, M., and Robertson, G. (2005). Easing program comprehension by sharing navigation data. In *Proc. of VL/HCC*, pages 241–248. IEEE.
- [Dempster et al., 1977] Dempster, A. P., Laird, N. M., and Rubin, D. B. (1977). Maximum likelihood from incomplete data via the em algorithm. *Journal of the royal statistical society. Series B (methodological)*, pages 1–38.
- [Dou et al., 2009] Dou, Z., Song, R., Nie, J.-Y., and Wen, J.-R. (2009). Using anchor texts with their hyperlink structure for web search. In *Proc. of SIGIR*, pages 227–234.
- [Duan et al., 2008] Duan, H., Cao, Y., Lin, C.-Y., and Yu, Y. (2008). Searching questions by identifying question topic and question focus. In *ACL*, pages 156–164.
- [Er et al., 2016] Er, M. J., Zhang, Y., Wang, N., and Pratama, M. (2016). Attention pooling-based convolutional neural network for sentence modelling. *Information Sciences*, 373:388–403.

- [Erkan and Radev, 2004] Erkan, G. and Radev, D. R. (2004). Lexrank: Graph-based lexical centrality as salience in text summarization. *Journal of Artificial Intelligence Research*, 22:457–479.
- [Figueroa and Neumann, 2016] Figueroa, A. and Neumann, G. (2016). Context-aware semantic classification of search queries for browsing community question-answering archives. *Knowledge-Based Systems*, 96:1–13.
- [Fleming et al., 2013] Fleming, S. D., Scaffidi, C., Piorkowski, D., Burnett, M., Bellamy, R., Lawrance, J., and Kwan, I. (2013). An information foraging theory perspective on tools for debugging, refactoring, and reuse tasks. *ACM TOSEM*, 22(2):14.
- [Foster et al., 2012] Foster, S. R., Griswold, W. G., and Lerner, S. (2012). Witchdoctor: Ide support for real-time auto-completion of refactorings. In *Proc. of ICSE*, pages 222–232.
- [Galenson et al., 2014] Galenson, J., Reames, P., Bodik, R., Hartmann, B., and Sen, K. (2014). Codehint: Dynamic and interactive synthesis of code snippets. In *Proc. of ICSE*, pages 653–663.
- [Ganesan et al., 2010] Ganesan, K., Zhai, C., and Han, J. (2010). Opinosis: a graph-based approach to abstractive summarization of highly redundant opinions. In *Proc. of COLING*, pages 340–348.
- [Genest and Lapalme, 2012] Genest, P.-E. and Lapalme, G. (2012). Fully abstractive approach to guided summarization. In *Proc. of ACL*, pages 354–358.
- [Goldstein et al., 1999] Goldstein, J., Kantrowitz, M., Mittal, V., and Carbonell, J. (1999). Summarizing text documents: sentence selection and evaluation metrics. In *Proc. of SIGIR*, pages 121–128.
- [Gomez et al., 2013] Gomez, C., Cleary, B., and Singer, L. (2013). A study of innovation diffusion through link sharing on stack overflow. In *Proc. of MSR*, pages 81–84.
- [Grbovic et al., 2015a] Grbovic, M., Djuric, N., Radosavljevic, V., Silvestri, F., and Bhamidipati, N. (2015a). Context-and content-aware embeddings for query rewriting in sponsored search. In *Proc. of SIGIR*, pages 383–392.

- [Grbovic et al., 2015b] Grbovic, M., Radosavljevic, V., Djuric, N., Bhamidipati, N., Savla, J., Bhagwan, V., and Sharp, D. (2015b). E-commerce in your inbox: Product recommendations at scale. In *Proc. of KDD*, pages 1809–1818.
- [Gu and Kim, 2015] Gu, X. and Kim, S. (2015). " what parts of your apps are loved by users?"(t). In *Proc. of ASE*, pages 760–770.
- [Gu et al., 2016] Gu, X., Zhang, H., Zhang, D., and Kim, S. (2016). Deep api learning. In *Proc. of FSE*. to appear.
- [Haiduc et al., 2010] Haiduc, S., Aponte, J., and Marcus, A. (2010). Supporting program comprehension with source code summarization. In *Proc. of ICSE*, volume 2, pages 223–226. IEEE.
- [Hall et al., 2009] Hall, M., Frank, E., Holmes, G., Pfahringer, B., Reutemann, P., and Witten, I. H. (2009). The weka data mining software: an update. *ACM SIGKDD explorations newsletter*, 11(1):10–18.
- [Harabagiu and Lacatusu, 2002] Harabagiu, S. M. and Lacatusu, F. (2002). Generating single and multi-document summaries with gistexter. In *Proc. of the Document Understanding Conferences*, pages 11–12.
- [He and Sun, 2015] He, K. and Sun, J. (2015). Convolutional neural networks at constrained time cost. In *Proc. of CVPR*, pages 5353–5360.
- [Hoffmann et al., 2007] Hoffmann, R., Fogarty, J., and Weld, D. S. (2007). Assieme: finding and leveraging implicit references in a web search interface for programmers. In *Proc. of UIST*. ACM.
- [Holmes and Murphy, 2005] Holmes, R. and Murphy, G. C. (2005). Using structural context to recommend source code examples. In *Proc. ICSE*, pages 117–125.
- [Holmes et al., 2005] Holmes, R., Walker, R. J., and Murphy, G. C. (2005). Strathcona example recommendation tool. In *Proc. of Software Engineering Notes*, volume 30, pages 237–240.

- [Hou et al., 2015] Hou, Y., Tan, C., Wang, X., Zhang, Y., Xu, J., and Chen, Q. (2015). Hitszicrc: Exploiting classification approach for answer selection in community question answering. In *Proc. of SemEval*, volume 15, pages 196–202.
- [Hua et al., 2015] Hua, W., Wang, Z., Wang, H., Zheng, K., and Zhou, X. (2015). Short text understanding through lexical-semantic analysis. In *Proc. of ICDE*, pages 495–506.
- [Jacobellis et al., 2014] Jacobellis, J., Meng, N., and Kim, M. (2014). Cookbook: In situ code completion using edit recipes learned from examples. In *Proc. of ICSE*, pages 584–587.
- [Jeon et al., 2005] Jeon, J., Croft, W. B., and Lee, J. H. (2005). Finding similar questions in large question and answer archives. In *Proc. of CIKM*, pages 84–90.
- [Ji et al., 2016] Ji, Z., Sun, A., Cong, G., and Han, J. (2016). Joint recognition and linking of fine-grained locations from tweets. In *Proc. of WWW*, pages 1271–1281.
- [Ji et al., 2012] Ji, Z., Xu, F., Wang, B., and He, B. (2012). Question-answer topic model for question retrieval in community question answering. In *Proc. of CIKM*, pages 2471–2474.
- [Jiang et al., 2017] Jiang, H., Zhang, J., Ren, Z., and Zhang, T. (2017). An unsupervised approach for discovering relevant tutorial fragments for apis. In *Proc. of ICSE*.
- [Kim et al., 2009] Kim, J., Lee, S., Hwang, S.-w., and Kim, S. (2009). Adding examples into java documents. In *Proc. of ASE*, pages 540–544.
- [Kim, 2014] Kim, Y. (2014). Convolutional neural networks for sentence classification. *arXiv preprint arXiv:1408.5882*.
- [Kingma and Ba, 2014] Kingma, D. and Ba, J. (2014). Adam: A method for stochastic optimization. *arXiv preprint arXiv:1412.6980*.
- [Klemmer et al., 2000] Klemmer, S. R., Sinha, A. K., Chen, J., Landay, J. A., Aboobaker, N., and Wang, A. (2000). Suede: a wizard of oz prototyping tool for speech user interfaces. In *Proc. of UIST*, pages 1–10. ACM.

- [Ko et al., 2005] Ko, A. J., Aung, H. H., and Myers, B. A. (2005). Eliciting design requirements for maintenance-oriented IDEs: a detailed study of corrective and perfective maintenance tasks. In *Proc. ICSE*, pages 126–135.
- [Ko et al., 2007] Ko, A. J., DeLine, R., and Venolia, G. (2007). Information needs in colocated software development teams. In *Proc. of ICSE*, pages 344–353.
- [Ko et al., 2006] Ko, A. J., Myers, B. A., Coblenz, M. J., and Aung, H. H. (2006). An exploratory study of how developers seek, relate, and collect relevant information during software maintenance tasks. *IEEE Transactions on software engineering*, 32(12):971–987.
- [Kokkinos and Margaritis, 2015] Kokkinos, Y. and Margaritis, K. G. (2015). Topology and simulations of a hierarchical markovian radial basis function neural network classifier. *Information Sciences*, 294:612–627.
- [Kramer, 1999] Kramer, D. (1999). Api documentation from source code comments: a case study of javadoc. In *Proc. of SIGDOC*, pages 147–153.
- [LaToza and Myers, 2010] LaToza, T. D. and Myers, B. A. (2010). Developers ask reachability questions. In *Proc. ICSE*, pages 185–194.
- [LaToza et al., 2006] LaToza, T. D., Venolia, G., and DeLine, R. (2006). Maintaining mental models: a study of developer work habits. In *Proc. of ICSE*, pages 492–501. ACM.
- [Lawrance et al., 2008] Lawrance, J., Bellamy, R., Burnett, M., and Rector, K. (2008). Using information scent to model the dynamic foraging behavior of programmers in maintenance tasks. In *Proc. of CHI*, pages 1323–1332. ACM.
- [Lawrance et al., 2013] Lawrance, J., Bogart, C., Burnett, M., Bellamy, R., Rector, K., and Fleming, S. D. (2013). How programmers debug, revisited: An information foraging theory perspective. *IEEE Trans. Soft. Eng.*, 39(2):197–215.
- [LeCun et al., 2015] LeCun, Y., Bengio, Y., and Hinton, G. (2015). Deep learning. *Nature*, 521(7553):436–444.

- [Lee et al., 2005] Lee, C.-S., Jian, Z.-W., and Huang, L.-K. (2005). A fuzzy ontology and its application to news summarization. *IEEE Transactions on Systems, Man, and Cybernetics, Part B (Cybernetics)*, 35(5):859–880.
- [Li et al., 2013] Li, H., Xing, Z., Peng, X., and Zhao, W. (2013). What help do developers seek, when and how? In *Proc. WCRE*, pages 142–151.
- [Li et al., 2015] Li, H., Zhao, X., Xing, Z., Bao, L., Peng, X., Gao, D., and Zhao, W. (2015). amassist: In-side ambient search of online programming resources. In *Proc. of SANER*, pages 390–398.
- [Li et al., 2018a] Li, J., Sun, A., and Xing, Z. (2018a). Learning to answer programming questions with software documentation through social context embedding. *Information Sciences*, 448 – 449:36–52.
- [Li et al., 2018b] Li, J., Xing, Z., and Kabir, A. (2018b). Leveraging official content and social context to recommend software documentation. *IEEE Transactions on Services Computing*, doi: 10.1109/TSC.2018.2812729.
- [Li et al., 2016] Li, J., Xing, Z., Ye, D., and Zhao, X. (2016). From discussion to wisdom: web resource recommendation for hyperlinks in stack overflow. In *Proc. of SAC*, pages 1127–1133. ACM.
- [Li et al., 2005] Li, Y., Lu, L., and Xuefeng, L. (2005). A hybrid collaborative filtering method for multiple-interests and multiple-content recommendation in e-commerce. *Expert Systems with Applications*, 28(1):67–77.
- [Lilliefors, 1967] Lilliefors, H. W. (1967). On the kolmogorov-smirnov test for normality with mean and variance unknown. *Journal of the American Statistical Association*, 62(318):399–402.
- [Lin, 2004] Lin, C.-Y. (2004). Rouge: A package for automatic evaluation of summaries. In *Proc. of ACL-04 workshop*, volume 8. Barcelona, Spain.
- [Lin and Hovy, 2003] Lin, C.-Y. and Hovy, E. (2003). Automatic evaluation of summaries using n-gram co-occurrence statistics. In *Proc. of NAACL HLT*, pages 71–78.

- [Lin et al., 2012] Lin, T., Pantel, P., Gamon, M., Kannan, A., and Fuxman, A. (2012). Active objects: Actions for entity-centric search. In *Proc. of WWW*, pages 589–598.
- [Linares-Vásquez et al., 2014] Linares-Vásquez, M., Bavota, G., Di Penta, M., Oliveto, R., and Poshyvanyk, D. (2014). How do api changes trigger stack overflow discussions? a study on the android sdk. In *Proc. of ICPC*, pages 83–94.
- [Linden et al., 2003] Linden, G., Smith, B., and York, J. (2003). Amazon. com recommendations: Item-to-item collaborative filtering. *Internet Computing, IEEE*, 7(1):76–80.
- [Liu, 2009] Liu, T.-Y. (2009). Learning to rank for information retrieval. *Foundations and Trends in Information Retrieval*, 3(3):225–331.
- [Lund and Burgess, 1996] Lund, K. and Burgess, C. (1996). Producing high-dimensional semantic spaces from lexical co-occurrence. *Behavior Research Methods, Instruments, & Computers*, 28(2):203–208.
- [MacQueen et al., 1967] MacQueen, J. et al. (1967). Some methods for classification and analysis of multivariate observations. In *Proc. of the fifth Berkeley symposium on mathematical statistics and probability*, volume 1, pages 281–297.
- [Marchionini, 2006] Marchionini, G. (2006). Exploratory search: from finding to understanding. *Communications of the ACM*, 49(4):41–46.
- [Matsuo and Ishizuka, 2004] Matsuo, Y. and Ishizuka, M. (2004). Keyword extraction from a single document using word co-occurrence statistical information. *Proc. of the International Journal on Artificial Intelligence Tools*, 13(01):157–169.
- [McMillan et al., 2011] McMillan, C., Grechanik, M., Poshyvanyk, D., Xie, Q., and Fu, C. (2011). Portfolio: finding relevant functions and their usage. In *Proc. ICSE*, pages 111–120.
- [Miao et al., 2010] Miao, Q., Li, Q., and Zeng, D. (2010). Fine-grained opinion mining by integrating multiple review sources. *Journal of the Association for Information Science and Technology*, 61(11):2288–2299.

- [Mikolov et al., 2013a] Mikolov, T., Chen, K., Corrado, G., and Dean, J. (2013a). Efficient estimation of word representations in vector space. *arXiv preprint arXiv:1301.3781*.
- [Mikolov et al., 2013b] Mikolov, T., Sutskever, I., Chen, K., Corrado, G. S., and Dean, J. (2013b). Distributed representations of words and phrases and their compositionality. In *Proc. of NIPS*, pages 3111–3119.
- [Miliaraki et al., 2015] Miliaraki, I., Blanco, R., and Lalmas, M. (2015). From selena gomez to marlon brando: Understanding explorative entity search. In *Proc. of WWW*, pages 765–775.
- [Miller, 1995] Miller, G. A. (1995). Wordnet: a lexical database for english. *Communications of the ACM*, 38(11):39–41.
- [Moody et al., 2010] Moody, M., Faulring, A., Stylos, J., and Myers, B. A. (2010). Calcite: Completing code completion for constructors using crowds. In *Proc. of VL/HCC*, pages 15–22. IEEE.
- [Moro et al., 2014] Moro, A., Raganato, A., and Navigli, R. (2014). Entity linking meets word sense disambiguation: a unified approach. *Transactions of the Association for Computational Linguistics*, 2:231–244.
- [Nadi et al., 2016] Nadi, S., Krüger, S., Mezini, M., and Bodden, E. (2016). Jumping through hoops: why do java developers struggle with cryptography apis? In *Proc. of ICSE*, pages 935–946.
- [Nair and Hinton, 2010] Nair, V. and Hinton, G. E. (2010). Rectified linear units improve restricted boltzmann machines. In *Proc. of ICML*, pages 807–814.
- [Nallapati et al., 2016] Nallapati, R., Zhou, B., dos Santos, C. N., Gülçehre, Ç., and Xiang, B. (2016). Abstractive text summarization using sequence-to-sequence rnns and beyond. In *Proc. of CoNLL*, pages 280–290.
- [Nasehi et al., 2012] Nasehi, S. M., Sillito, J., Maurer, F., and Burns, C. (2012). What makes a good code example?: A study of programming q&a in stackoverflow. In *Proc. of ICSM*, pages 25–34.

- [Nassif et al., 2016] Nassif, H., Mohtarami, M., and Glass, J. (2016). Learning semantic relatedness in community question answering using neural models. *Proc. of ACL*, page 137.
- [Nguyen et al., 2012] Nguyen, A. T., Nguyen, H. A., Nguyen, T. T., and Nguyen, T. N. (2012). Grapacc: a graph-based pattern-oriented, context-sensitive code completion tool. In *Proc. of ICSE*, pages 1407–1410.
- [Nicosia et al., 2015] Nicosia, M., Filice, S., Barrón-Cedeno, A., Saleh, I., Mubarak, H., Gao, W., Nakov, P., Da San Martino, G., Moschitti, A., Darwish, K., et al. (2015). Qcri: Answer selection for community question answering experiments for arabic and english. In *SemEval*, volume 15, pages 203–209.
- [Niu et al., 2016] Niu, H., Keivanloo, I., and Zou, Y. (2016). Learning to rank code examples for code search engines. *Empirical Software Engineering*, pages 1–33.
- [Palomera and Figueroa, 2017] Palomera, D. and Figueroa, A. (2017). Leveraging linguistic traits and semi-supervised learning to single out informational content across how-to community question-answering archives. *Information Sciences*, 381:20–32.
- [Park et al., 2008] Park, Y., Patwardhan, S., Visweswariah, K., and Gates, S. C. (2008). An empirical analysis of word error rate and keyword error rate. In *Proc. of INTER-SPEECH*, pages 2070–2073.
- [Parnin and Gorg, 2006] Parnin, C. and Gorg, C. (2006). Building usage contexts during program comprehension. In *Proc. of ICPC*, pages 13–22. IEEE.
- [Parnin and Rugaber, 2012] Parnin, C. and Rugaber, S. (2012). Programmer information needs after memory failure. In *Proc. ICPC*, pages 123–132.
- [Parnin et al., 2012] Parnin, C., Treude, C., Grammel, L., and Storey, M.-A. (2012). Crowd documentation: Exploring the coverage and the dynamics of api discussions on stack overflow. *Georgia Institute of Technology, Tech. Rep.*

- [Paulus et al., 2017] Paulus, R., Xiong, C., and Socher, R. (2017). A deep reinforced model for abstractive summarization. *CoRR*, abs/1705.04304.
- [Perpich et al., 1997] Perpich, J. M., Perry, D. E., Porter, A. A., Votta, L. G., and Wade, M. W. (1997). Anywhere, anytime code inspections: using the web to remove inspection bottlenecks in large-scale software development. In *Proc. of ICSE*.
- [Petrosyan et al., 2015] Petrosyan, G., Robillard, M. P., and De Mori, R. (2015). Discovering information explaining api types using text classification. In *Proc. of ICSE*, pages 869–879.
- [Pletcher and Hou, 2009] Pletcher, D. M. and Hou, D. (2009). Bcc: Enhancing code completion for better api usability. In *Proc. of ICSM*, pages 393–394.
- [Ponzanelli et al., 2013a] Ponzanelli, L., Bacchelli, A., and Lanza, M. (2013a). Leveraging crowd knowledge for software comprehension and development. In *Proc. of ECSME*, pages 57–66.
- [Ponzanelli et al., 2013b] Ponzanelli, L., Bacchelli, A., and Lanza, M. (2013b). Seahawk: Stack overflow in the ide. In *Proc. of ICSE*, pages 1295–1298.
- [Ponzanelli et al., 2014a] Ponzanelli, L., Bavota, G., Di Penta, M., Oliveto, R., and Lanza, M. (2014a). Mining stackoverflow to turn the ide into a self-confident programming prompter. In *Proc. of ICMSE*, pages 102–111.
- [Ponzanelli et al., 2014b] Ponzanelli, L., Bavota, G., Di Penta, M., Oliveto, R., and Lanza, M. (2014b). Prompter: A self-confident recommender system. In *Proc. of ICSME*, pages 577–580.
- [Quinn, 1994] Quinn, M. J. (1994). *Parallel computing: theory and practice*. McGraw-Hill, Inc.
- [Quoc and Le, 2007] Quoc, C. and Le, V. (2007). Learning to rank with nonsmooth cost functions. *Proc. of NIPS*, 19:193–200.
- [Raghothaman et al., 2016] Raghothaman, M., Wei, Y., and Hamadi, Y. (2016). Swim: Synthesizing what i mean. In *Proc. of ICSE*.

- [Rahman et al., 2016] Rahman, M. M., Roy, C. K., and Lo, D. (2016). Rack: Automatic api recommendation using crowdsourced knowledge. In *Proc. of SANER*, volume 1, pages 349–359.
- [Ren et al., 2017] Ren, P., Chen, Z., Ren, Z., Wei, F., Ma, J., and de Rijke, M. (2017). Leveraging contextual sentence relations for extractive summarization using a neural attention model. In *Proc. of SIGIR*, pages 95–104. ACM.
- [Robertson et al., 1995] Robertson, S., Walker, S., Jones, S., Hancock-Beaulieu, M. M., and Gatford, M. (1995). Okapi at trec3. In *Overview of the Third Text REtrieval Conference (TREC3)*, page 109126. Gaithersburg, MD: NIST.
- [Robillard and Deline, 2011] Robillard, M. P. and Deline, R. (2011). A field study of api learning obstacles. *Empirical Software Engineering*, 16(6):703–732.
- [Robillard et al., 2014] Robillard, M. P., Maalej, W., Walker, R. J., and Zimmermann, T. (2014). *Recommendation systems in software engineering*. Springer Science & Business.
- [Robillard et al., 2010] Robillard, M. P., Walker, R. J., and Zimmermann, T. (2010). Recommendation systems for software engineering. *Software, IEEE*, 27(4):80–86.
- [Sahavechaphan and Claypool, 2006] Sahavechaphan, N. and Claypool, K. (2006). Xsnippet: mining for sample code. *ACM Sigplan Notices*, 41(10):413–430.
- [Sakai et al., 2011] Sakai, T., Ishikawa, D., Kando, N., Seki, Y., Kuriyama, K., and Lin, C.-Y. (2011). Using graded-relevance metrics for evaluating community qa answer selection. In *Proc. of WSDM*, pages 187–196.
- [Sakai and Nogami, 2009] Sakai, T. and Nogami, K. (2009). Serendipitous search via wikipedia: a query log analysis. In *Proc. of SIGIR*, pages 780–781.
- [San Pedro and Karatzoglou, 2014] San Pedro, J. and Karatzoglou, A. (2014). Question recommendation for collaborative question answering systems with ranklda. In *Proc. of RecSys*, pages 193–200.
- [Sanderson and Fisher, 1994] Sanderson, P. M. and Fisher, C. (1994). Exploratory sequential data analysis: Foundations. *Human–Computer Interaction*, 9(3-4):251–317.

- [Sarwar et al., 2001] Sarwar, B., Karypis, G., Konstan, J., and Riedl, J. (2001). Item-based collaborative filtering recommendation algorithms. In *Proc. of WWW*, pages 285–295.
- [Schmidhuber, 2015] Schmidhuber, J. (2015). Deep learning in neural networks: An overview. *Neural Networks*, 61:85–117.
- [Seebach, 2012] Seebach, C. (2012). Searching for answers–knowledge exchange through social media in organizations. In *Proc. of HICSS*, pages 3908–3917.
- [Serva et al., 2015] Serva, R., Senzer, Z. R., Pollock, L., and Vijay-Shanker, K. (2015). Automatically mining negative code examples from software developer q & a forums. In *Proc. of ASE*, pages 115–122.
- [Severyn and Moschitti, 2015] Severyn, A. and Moschitti, A. (2015). Learning to rank short text pairs with convolutional deep neural networks. In *SIGIR*, pages 373–382.
- [Severyn and Moschitti, 2016] Severyn, A. and Moschitti, A. (2016). Modeling relational information in question-answer pairs with convolutional neural networks. *arXiv preprint arXiv:1604.01178*.
- [Shen et al., 2015] Shen, W., Wang, J., and Han, J. (2015). Entity linking with a knowledge base: Issues, techniques, and solutions. *IEEE TKDE*, 27(2):443–460.
- [Sillito et al., 2006] Sillito, J., Murphy, G. C., and De Volder, K. (2006). Questions programmers ask during software evolution tasks. In *Proc. of FSE*, pages 23–34.
- [Sillito et al., 2008] Sillito, J., Murphy, G. C., and De Volder, K. (2008). Asking and answering questions during a programming change task. *IEEE Trans. Soft. Eng.*, 34(4):434–451.
- [Singh and Simperl, 2016] Singh, P. and Simperl, E. (2016). Using semantics to search answers for unanswered questions in q&a forums. In *Proc. of WWW*, pages 699–706.
- [Sinha et al., 2006] Sinha, S. N., Frahm, J.-M., Pollefeys, M., and Genc, Y. (2006). Gpu-based video feature tracking and matching. In *Proc. of EDGE*, volume 278, page 4321.

- [Stylos et al., 2009] Stylos, J., Faulring, A., Yang, Z., and Myers, B. A. (2009). Improving api documentation using api usage information. In *Proc. of VL/HCC*, pages 119–126.
- [Stylos and Myers, 2006] Stylos, J. and Myers, B. A. (2006). Mica: A web-search tool for finding api components and examples. In *Proc. of VL/HCC*.
- [Subramanian et al., 2014] Subramanian, S., Inozemtseva, L., and Holmes, R. (2014). Live api documentation. In *Proc. of ICSE*, pages 643–652.
- [Sun et al., 2005] Sun, R., Cui, H., Li, K., Kan, M.-Y., and Chua, T.-S. (2005). Dependency relation matching for answer selection. In *Proc. of SIGIR*, pages 651–652.
- [Surdeanu et al., 2008] Surdeanu, M., Ciaramita, M., and Zaragoza, H. (2008). Learning to rank answers on large online qa collections. In *Proc. of ACL*, volume 8, pages 719–727.
- [Tan et al., 2017] Tan, J., Wan, X., and Xiao, J. (2017). Abstractive document summarization with a graph-based attentional neural model. In *Proc. of ACL*, pages 1171–1181.
- [Tan et al., 2016] Tan, M., dos Santos, C., Xiang, B., and Zhou, B. (2016). Improved representation learning for question answer matching. In *Proc. of ACL*, pages 464–473.
- [Tang et al., 2014] Tang, D., Wei, F., Yang, N., Zhou, M., Liu, T., and Qin, B. (2014). Learning sentiment-specific word embedding for twitter sentiment classification. In *Proc. of ACL*, volume 1, pages 1555–1565.
- [Tenopir and King, 2004] Tenopir, C. and King, D. W. (2004). *Communication patterns of engineers*. John Wiley & Sons.
- [Thummalapenta and Xie, 2007] Thummalapenta, S. and Xie, T. (2007). Parseweb: a programmer assistant for reusing open source code on the web. In *Proc. ASE*, pages 204–213.

- [Tjong Kim Sang and De Meulder, 2003] Tjong Kim Sang, E. F. and De Meulder, F. (2003). Introduction to the conll-2003 shared task: Language-independent named entity recognition. In *Proc. of HLT-NAACL*, pages 142–147.
- [Toba et al., 2014] Toba, H., Ming, Z.-Y., Adriani, M., and Chua, T.-S. (2014). Discovering high quality answers in community question answering archives using a hierarchy of classifiers. *Information Sciences*, 261:101–115.
- [Treude et al., 2011] Treude, C., Barzilay, O., and Storey, M.-A. (2011). How do programmers ask and answer questions on the web?: Nier track. In *Proc. of ICSE*, pages 804–807.
- [Treude and Robillard,] Treude, C. and Robillard, M. Augmenting api documentation with insights from stack overflow. In *Proc. of ICSE*.
- [Treude and Robillard, 2016] Treude, C. and Robillard, M. P. (2016). Augmenting api documentation with insights from stack overflow. In *Proc. of ICSE*, pages 392–403.
- [Turian et al., 2010] Turian, J., Ratinov, L., and Bengio, Y. (2010). Word representations: a simple and general method for semi-supervised learning. In *Proc. of ACL*, pages 384–394.
- [Van Nguyen et al., 2016] Van Nguyen, T., Nguyen, A. T., and Nguyen, T. N. (2016). Characterizing api elements in software documentation with vector representation. In *Proc. of ICSE*, pages 749–751.
- [Von Mayrhauser and Vans, 1995] Von Mayrhauser, A. and Vans, A. M. (1995). Program comprehension during software maintenance and evolution. *Computer*, 28(8):44–55.
- [Vu et al., 2015] Vu, P. M., Nguyen, T. T., Pham, H. V., and Nguyen, T. T. (2015). Mining user opinions in mobile app reviews: A keyword-based approach (t). In *Proc. of ASE*, pages 749–759.
- [Vu et al., 2016] Vu, P. M., Pham, H. V., Nguyen, T. T., et al. (2016). Phrase-based extraction of user opinions in mobile app reviews. In *Proc. of ASE*, pages 726–731.

- [Wang et al., 2012] Wang, D., Zhu, S., Li, T., and Gong, Y. (2012). Comparative document summarization via discriminative sentence selection. *ACM TKDD*, 6(3):12.
- [Wang et al., 2006] Wang, J., De Vries, A. P., and Reinders, M. J. (2006). Unifying user-based and item-based collaborative filtering approaches by similarity fusion. In *Proc. of SIGIR*, pages 501–508.
- [Wang et al., 2013] Wang, S., Lo, D., and Jiang, L. (2013). An empirical study on developer interactions in stackoverflow. In *Proc. of SAC*, pages 1019–1024.
- [Wang et al., 2014] Wang, S., Lo, D., Vasilescu, B., and Serebrenik, A. (2014). Entagrec: an enhanced tag recommendation system for software information sites. In *Proc. of ICSME*, pages 291–300.
- [Wettel and Lanza, 2007] Wettel, R. and Lanza, M. (2007). Program comprehension through software habitability. In *Proc. of ICPC*, pages 231–240. IEEE.
- [White and Roth, 2009] White, R. W. and Roth, R. A. (2009). Exploratory search: Beyond the query-response paradigm. *Synthesis Lectures on Information Concepts, Retrieval, and Services*, 1(1):1–98.
- [Whittaker and Poore, 1993] Whittaker, J. A. and Poore, J. H. (1993). Markov analysis of software specifications. *ACM TOSEM*, 2(1):93–106.
- [Wisniewski et al., 2015] Wisniewski, P., Xu, H., Lipford, H., and Bello-Ogunu, E. (2015). Facebook apps and tagging: The trade-off between personal privacy and engaging with friends. *Journal of the Association for Information Science and Technology*, 66(9):1883–1896.
- [Wu et al., 2010] Wu, Q., Burges, C. J., Svore, K. M., and Gao, J. (2010). Adapting boosting for information retrieval measures. *Information Retrieval*, 13(3):254–270.
- [Xue et al., 2008] Xue, X., Jeon, J., and Croft, W. B. (2008). Retrieval models for question and answer archives. In *Proc. of SIGIR*, pages 475–482.

- [Yan et al., 2016] Yan, R., Song, Y., and Wu, H. (2016). Learning to respond with deep neural networks for retrieval-based human-computer conversation system. In *Proc. of SIGIR*, pages 55–64.
- [Yao et al., 2015] Yao, Y., Tong, H., Xie, T., Akoglu, L., Xu, F., and Lu, J. (2015). Detecting high-quality posts in community question answering sites. *Information Sciences*, 302:70–82.
- [Ye et al., 2016a] Ye, D., Xing, Z., Foo, C. Y., Ang, Z. Q., Li, J., and Kapre, N. (2016a). Software-specific named entity recognition in software engineering social content. In *Proc. of SANER*, volume 1, pages 90–101.
- [Ye et al., 2016b] Ye, D., Xing, Z., Foo, C. Y., Li, J., and Kapre, N. (2016b). Learning to extract api mentions from informal natural language discussions. In *Proc. of ICSME*, pages 389–399.
- [Ye et al., 2016c] Ye, D., Xing, Z., Li, J., and Kapre, N. (2016c). Software-specific part-of-speech tagging: an experimental study on stack overflow. In *Proc. of SAC*, pages 1378–1385. ACM.
- [Yen et al., 2013] Yen, S.-J., Wu, Y.-C., Yang, J.-C., Lee, Y.-S., Lee, C.-J., and Liu, J.-J. (2013). A support vector machine-based context-ranking model for question answering. *Information Sciences*, 224:77–87.
- [Yu et al., 2014] Yu, L., Hermann, K. M., Blunsom, P., and Pulman, S. (2014). Deep learning for answer sentence selection. *arXiv preprint arXiv:1412.1632*.
- [Yuan et al., 2014] Yuan, Q., Cong, G., and Sun, A. (2014). Graph-based point-of-interest recommendation with geographical and temporal influences. In *Proc. of CIKM*, pages 659–668.
- [Zagalsky et al., 2012] Zagalsky, A., Barzilay, O., and Yehudai, A. (2012). Example overflow: Using social media for code recommendation. In *Proc. of the Third International Workshop on Recommendation Systems for Software Engineering*, pages 38–42.

- [Zhong et al., 2009] Zhong, H., Xie, T., Zhang, L., Pei, J., and Mei, H. (2009). Mapo: Mining and recommending api usage patterns. In *ECOOOP 2009–Object-Oriented Programming*, pages 318–343. Springer.
- [Zhou et al., 2011] Zhou, G., Cai, L., Zhao, J., and Liu, K. (2011). Phrase-based translation model for question retrieval in community question answer archives. In *Proc. of ACL*, pages 653–662.
- [Zhou et al., 2015] Zhou, G., He, T., Zhao, J., and Hu, P. (2015). Learning continuous word embedding with metadata for question retrieval in community question answering. In *Proc. of ACL*, pages 250–259.
- [Zhou et al., 2013] Zhou, G., Liu, Y., Liu, F., Zeng, D., and Zhao, J. (2013). Improving question retrieval in community question answering using world knowledge. In *Proc. of IJCAI*, volume 13, pages 2239–2245.
- [Zhou et al., 2016] Zhou, G., Zhou, Y., He, T., and Wu, W. (2016). Learning semantic representation with neural networks for community question answering retrieval. *Knowledge-Based Systems*, 93:75–83.
- [Zhou and Mockus, 2011] Zhou, M. and Mockus, A. (2011). Does the initial environment impact the future of developers? In *Proc. of ICSE*, pages 271–280. ACM.
- [Zou et al., 2015] Zou, Y., Ye, T., Lu, Y., Mylopoulos, J., and Zhang, L. (2015). Learning to rank for question-oriented software text retrieval (t). In *Proc. of ASE*, pages 1–11.