

SUPPORTING SOURCE CODE SEARCH WITH CONTEXT-AWARE  
AND SEMANTICS-DRIVEN QUERY REFORMULATION

A Thesis Submitted to the  
College of Graduate and Postdoctoral Studies  
in Partial Fulfillment of the Requirements  
for the degree of Doctor of Philosophy  
in the Department of Computer Science  
University of Saskatchewan  
Saskatoon

By  
Mohammad Masudur Rahman

©Mohammad Masudur Rahman, September/2019. All rights reserved.

## PERMISSION TO USE

In presenting this thesis in partial fulfilment of the requirements for a Postgraduate degree from the University of Saskatchewan, I agree that the Libraries of this University may make it freely available for inspection. I further agree that permission for copying of this thesis in any manner, in whole or in part, for scholarly purposes may be granted by the professor or professors who supervised my thesis work or, in their absence, by the Head of the Department or the Dean of the College in which my thesis work was done. It is understood that any copying or publication or use of this thesis or parts thereof for financial gain shall not be allowed without my written permission. It is also understood that due recognition shall be given to me and to the University of Saskatchewan in any scholarly use which may be made of any material in my thesis.

Requests for permission to copy or to make other use of material in this thesis in whole or part should be addressed to:

Head of the Department of Computer Science  
176 Thorvaldson Building  
110 Science Place  
University of Saskatchewan  
Saskatoon, Saskatchewan  
Canada  
S7N 5C9

# ABSTRACT

Software bugs and failures cost trillions of dollars every year, and could even lead to deadly accidents (e.g., Therac-25 accident). During maintenance, software developers fix numerous bugs and implement hundreds of new features by making necessary changes to the existing software code. Once an issue report (e.g., bug report, change request) is assigned to a developer, she chooses a few important keywords from the report as a search query, and then attempts to find out the exact locations in the software code that need to be either repaired or enhanced. As a part of this maintenance, developers also often select ad hoc queries on the fly, and attempt to locate the reusable code from the Internet that could assist them either in bug fixing or in feature implementation. Unfortunately, even the experienced developers often fail to construct the right search queries. Even if the developers come up with a few ad hoc queries, most of them require frequent modifications which cost significant development time and efforts. Thus, construction of an appropriate query for localizing the software bugs, programming concepts or even the reusable code is a major challenge. In this thesis, we overcome this query construction challenge with six studies, and develop a novel, effective code search solution (BugDoctor) that assists the developers in localizing the software code of interest (e.g., bugs, concepts and reusable code) during software maintenance. In particular, we reformulate a given search query (1) by designing novel keyword selection algorithms (e.g., CodeRank) that outperform the traditional alternatives (e.g., TF-IDF), (2) by leveraging the bug report quality paradigm and source document structures which were previously overlooked and (3) by exploiting the crowd knowledge and word semantics derived from Stack Overflow Q&A site, which were previously untapped. Our experiment using 5000+ search queries (bug reports, change requests, and ad hoc queries) suggests that our proposed approach can improve the given queries significantly through automated query reformulations. Comparison with 10+ existing studies on bug localization, concept location and Internet-scale code search suggests that our approach can outperform the state-of-the-art approaches with a significant margin.

## ACKNOWLEDGEMENTS

First, I thank the Almighty, the most gracious and the most merciful, who granted me all the abilities to carry out this work. Then I would like to express my heartiest gratitude to my advisor Dr. Chanchal K. Roy for his constant guidance, advice, critical insights, positive encouragements and extraordinary patience during this thesis work. He is definitely a great mentor who can bring out the best in a student. This work would have been impossible without his supports.

I would like to thank Dr. Denys Poshyvanyk, Dr. Andrew Grosvenor, Dr. Ian Stavness, Dr. Natalia Stakhanova, Dr. Debajyoti Mondal and Dr. Banani Roy for their willingness to take part in the advisement and evaluation of my thesis work. I would also like to thank them for their valuable time, useful suggestions and critical insights. Their comments helped improve my thesis significantly.

I would like to convey my greatest love and gratitude to my beloved wife, Shamima Yeasmin, and my lovely daughter, Anisha. They are the love and inspirations of my life. Shamima always stayed with me in ease and hardship, inspired me constantly, and helped me with ideas and suggestions in this work. Anisha helped me see the life in a new light with her heavenly innocence, unforgettable smiles and constant babbling.

I would like to express my deepest love to my mother Morium Begum and my father Md. Sadiqur Rahman who brought me to this world. Their endless sacrifice, unconditional love and constant well wishes have made me reach this stage of my life. I would also like to thank my mother-in-law Mrs. Rezia Khatun and father-in-law Md. Shamsul Islam for their constant well wishes and inspirations in this thesis work. My siblings – Asad, Mamun and Sayed, and in-laws – Masum, Mamun, Maruf, Shefa, Rabeya, Farzana, Dipa, and Sharmin have always inspired me in completing my thesis work, and I thank all of them.

I specially thank all the members of Software Research Lab with whom I have had the opportunity to grow as a researcher. In particular, I would like to thank Dr. Manishankar Mondal, Dr. Md. Saidur Rahman, Dr. Muhammad Asaduzzaman, Dr. Jeffrey Svajlenko, Dr. Fahim Zibran, Judith, Farouq, Mostaeen, Saikat, Amit, Nafi, Joy, Khaled, Rayhan, Avijit, Nadim, Rodrigo and Hamid. I am grateful to the University of Saskatchewan and its Department of Computer Science for their generous financial supports through scholarships, awards and bursaries. I am also grateful to NSERC for the prestigious Industry Engage grant. All these supports helped me concentrate deeply in my thesis work.

I thank all the anonymous reviewers and editors for their valuable comments and suggestions in improving the research papers produced from this thesis. I would also like to thank my research collaborators – Dr. David Lo, Dr. Raula G. Kula and Dr. Iman Keivanloo – for their collaborations.

I would like to thank all of my friends and staff members from the Department of Computer Science who have helped me reach this stage with time, efforts and suggestions. In particular, I would like to thank Dr. Rafizul Haque, Priyasree Bhowmik, Farhad Maleki, Kimberly Mackay, Varun Gaur, Kiemute Oyibo, Rasam Bin Hossain, Sowgat Ibne Mahmud, Sami Uddin, Aminul Islam, Nazifa Azam Khan and Gwen Lancaster, Shakiba Jalal, Sophie Findlay and Heather Webb.

I dedicate this thesis to my mother Mrs. Morium Begum and my father Md. Sadiqur Rahman whose inspirations help me accomplish every goal of my life.

# CONTENTS

<b>Permission to Use</b>	<b>i</b>
<b>Abstract</b>	<b>ii</b>
<b>Acknowledgements</b>	<b>iii</b>
<b>Contents</b>	<b>v</b>
<b>List of Tables</b>	<b>x</b>
<b>List of Figures</b>	<b>xii</b>
<b>List of Abbreviations</b>	<b>xv</b>
<b>1 Introduction</b>	<b>1</b>
1.1 Motivation . . . . .	1
1.2 Problem Statement . . . . .	2
1.3 Our Contribution . . . . .	5
1.3.1 Concept Location . . . . .	5
1.3.2 Bug Localization . . . . .	5
1.3.3 Internet-scale Code Search . . . . .	6
1.4 Related Publications . . . . .	9
1.5 Outline of the Thesis . . . . .	12
<b>2 Background</b>	<b>14</b>
2.1 Automated Query Reformulation . . . . .	14
2.2 Types of Query Reformulation . . . . .	14
2.2.1 Query Expansion . . . . .	15
2.2.2 Query Reduction . . . . .	15
2.2.3 Query Replacement . . . . .	15
2.3 Working Contexts of Query Reformulation . . . . .	15
2.3.1 Local Code Search . . . . .	16
2.3.2 Internet-Scale Code Search . . . . .	16
2.4 Steps of Automated Query Reformulation . . . . .	17
2.4.1 Query Feedback Collection . . . . .	17
2.4.2 Candidate Keyword Selection . . . . .	18
2.4.3 Reformulation of a Search Query . . . . .	18
2.5 Term Weighting . . . . .	18
2.5.1 TF-IDF . . . . .	19
2.5.2 TextRank & POSRank . . . . .	19
2.6 Implications of Automated Query Reformulation . . . . .	20
2.6.1 Benefits of Query Reformulation . . . . .	20
2.6.2 Costs of Query Reformulation . . . . .	20
2.7 Word Embeddings . . . . .	21
2.8 Cosine Similarity . . . . .	21
2.9 Summary . . . . .	22
<b>3 Search Query Reformulation for Concept Location using Graph-Based Term Weighting</b>	<b>23</b>
3.1 Introduction . . . . .	23
3.2 Motivating Example . . . . .	26

3.3	STRICT: Automated Search Query Suggestion from a Change Request for Concept Location	27
3.3.1	Data Collection	28
3.3.2	Text Preprocessing	28
3.3.3	Text Graph Development	28
3.3.4	TextRank (TR) Calculation	29
3.3.5	POSRank (POSR) Calculation	30
3.3.6	Weighted K-Core Calculation	31
3.3.7	Term Ranking and Candidate Query Selection	31
3.3.8	Best Query Suggestion with Machine Learning	33
3.3.9	A Working Example	35
3.4	Experiment	35
3.4.1	Experimental Dataset	37
3.4.2	Search Engine	37
3.4.3	Performance Metrics	38
3.4.4	Evaluation of STRICT	39
3.4.5	Comparison with Existing Techniques	48
3.4.6	Evaluation of Working Prototype	54
3.5	Threats to Validity	56
3.6	Related Work	58
3.6.1	Search Query Suggestion & Reformulation	58
3.6.2	Code Search Algorithm	59
3.7	Summary	60
<b>4</b>	<b>Search Query Reformulation for Concept Location using CodeRank and Source Document Structures</b>	<b>61</b>
4.1	Introduction	61
4.2	ACER: Automated Query Reformulation with CodeRank and Document Structures for Concept Location	65
4.2.1	Pseudo-relevance Feedback	65
4.2.2	Source Token Selection for Query Reformulation	66
4.2.3	Source Code Preprocessing	67
4.2.4	Source Term Graph Development	67
4.2.5	CodeRank Calculation	68
4.2.6	Suggestion of the Best Query Reformulation	69
4.3	Experiment	72
4.3.1	Experimental Dataset	72
4.3.2	Corpus Indexing & Source Code Search	73
4.3.3	Performance Metrics	74
4.3.4	Evaluation of ACER and CodeRank	74
4.3.5	Comparison with Existing Approaches	81
4.4	Threats to Validity	82
4.5	Related Work	83
4.6	Summary	84
<b>5</b>	<b>Search Query Reformulation for Bug Localization using Report Quality Dynamics &amp; Graph-Based Term Weighting</b>	<b>85</b>
5.1	Introduction	85
5.2	BLIZZARD: Automated Query Suggestion using Report Quality Dynamics and Term Weighting for Bug Localization	88
5.2.1	Bug Report Classification	88
5.2.2	Query Reformulation	89
5.2.3	Bug Localization	94
5.3	Experiment	95
5.3.1	Experimental Dataset	95

5.3.2	Performance Metrics . . . . .	96
5.3.3	Experimental Results . . . . .	97
5.3.4	Comparison with Existing Techniques . . . . .	101
5.4	Threats to Validity . . . . .	107
5.5	Related Work . . . . .	107
5.6	Summary . . . . .	108
<b>6</b>	<b>Search Query Reformulation for Bug Localization using Word Semantics &amp; Clustering Tendency Analysis</b>	<b>110</b>
6.1	Introduction . . . . .	110
6.2	Motivating Example . . . . .	113
6.3	BLADER: Automated Query Reformulation using Word Semantics & Clustering Tendency Analysis for Bug Localization . . . . .	114
6.3.1	Construction of a Semantic Hyperspace from Stack Overflow Q&A Threads . . . . .	114
6.3.2	Automated Search Query Reformulation with Semantic Hyperspace, Clustering Tendency & Machine Learning . . . . .	117
6.3.3	Bug Localization . . . . .	120
6.4	Experiment . . . . .	120
6.4.1	Experimental Dataset . . . . .	120
6.4.2	Performance Metrics . . . . .	121
6.4.3	Evaluation of BLADER . . . . .	122
6.4.4	Comparison with Existing Techniques . . . . .	125
6.5	Threats to Validity . . . . .	130
6.6	Related Work . . . . .	131
6.7	Summary . . . . .	132
<b>7</b>	<b>Search Query Reformulation for Internet-scale Code Search using Crowdsourced Knowledge</b>	<b>134</b>
7.1	Introduction . . . . .	134
7.2	Exploratory Study . . . . .	138
7.2.1	Data Collection . . . . .	138
7.2.2	API Class Name Extraction . . . . .	140
7.2.3	Answering <b>RQ<sub>1</sub></b> : Use of APIs in the accepted answers of Stack Overflow . . . . .	141
7.2.4	Answering <b>RQ<sub>2</sub></b> : Coverage of API classes in the accepted answers from Stack Overflow Q & A site . . . . .	142
7.2.5	Answering <b>RQ<sub>3</sub></b> : Presence of code search keywords in the title of questions from Stack Overflow . . . . .	144
7.3	RACK: Automated Query Reformulation for Internet-scale Code Search using Crowdsourced Knowledge . . . . .	148
7.3.1	Construction of NL Token-API Mapping Database . . . . .	149
7.3.2	API Relevance Ranking & Reformulation of the NL-Query . . . . .	150
7.4	Experiment . . . . .	156
7.4.1	Experimental Dataset . . . . .	156
7.4.2	Performance Metrics . . . . .	159
7.4.3	Evaluation Scenarios . . . . .	161
7.4.4	Statistical Significance Tests . . . . .	161
7.4.5	Matching of Suggested APIs with Goldset APIs . . . . .	161
7.4.6	Answering <b>RQ<sub>4</sub></b> : How does the proposed technique perform in suggesting relevant APIs for a code search query? . . . . .	162
7.4.7	Answering <b>RQ<sub>5</sub></b> : How effective are the proposed heuristics—KAC, KPAC and KKC—in capturing the relevant API classes for a query? . . . . .	164
7.4.8	Answering <b>RQ<sub>6</sub></b> : Does an appropriate subset of the query keywords perform better than the whole query in retrieving the relevant API classes? . . . . .	166



7.4.9	Answering <b>RQ<sub>7</sub></b> : How do the heuristic weights (i.e., $\alpha, \beta$ ) and threshold settings (i.e., $\gamma, \delta$ ) influence the performance of our technique? . . . . .	167
7.4.10	Answering <b>RQ<sub>8</sub></b> : Can RACK outperform the state-of-the-art techniques in suggesting relevant API classes for a given set of queries? . . . . .	169
7.4.11	Answering <b>RQ<sub>9</sub></b> : Can RACK significantly improve the natural language queries in terms of relevant code retrieval performance? . . . . .	172
7.4.12	Answering <b>RQ<sub>10</sub></b> : Can RACK outperform the state-of-the-art techniques in improving the natural language queries intended for code search? . . . . .	176
7.4.13	Answering <b>RQ<sub>11</sub></b> : How does RACK perform compared to the popular web search engines and code search engines? . . . . .	181
7.5	Threats to Validity . . . . .	185
7.5.1	Threats to Internal Validity . . . . .	185
7.5.2	Threats to External Validity . . . . .	185
7.5.3	Threats to Construct Validity . . . . .	186
7.5.4	Threats to Statistical Conclusion Validity . . . . .	186
7.6	Related Work . . . . .	186
7.6.1	API Recommendation . . . . .	186
7.6.2	API Usage Pattern Recommendation . . . . .	187
7.6.3	Query Reformulation for Code Search . . . . .	187
7.6.4	Crowdsourced Knowledge Mining . . . . .	188
7.7	Summary . . . . .	189
<b>8</b>	<b>Search Query Reformulation for Internet-scale Code Search using Word Semantics</b>	<b>191</b>
8.1	Introduction . . . . .	191
8.2	NLP2API: Automated Query Reformulation using Word Semantics & Crowd Knowledge for Internet-scale Code Search . . . . .	194
8.2.1	Development of Candidate API Lists . . . . .	194
8.2.2	Borda Score Calculation . . . . .	197
8.2.3	Query-API Semantic Proximity Analysis . . . . .	199
8.2.4	API Class Relevance Ranking & Query Reformulation . . . . .	200
8.3	Experiment . . . . .	200
8.3.1	Experimental Dataset . . . . .	201
8.3.2	Performance Metrics . . . . .	201
8.3.3	Evaluation of NLP2API: Relevant API Class Suggestion . . . . .	202
8.3.4	Evaluation of NLP2API: Query Reformulation . . . . .	205
8.4	Threats to Validity . . . . .	209
8.5	Related Work . . . . .	210
8.6	Summary . . . . .	211
<b>9</b>	<b>Conclusion</b>	<b>213</b>
9.1	Concluding Remarks . . . . .	213
9.2	Future Work . . . . .	216
9.2.1	Promises of Keyword Selection Algorithms in IR-Based Bug Localization . . . . .	217
9.2.2	Promises of Genetic Algorithms in IR-Based Bug Localization . . . . .	217
9.2.3	Improving Term Weighting Algorithms with Useful Term Contexts . . . . .	218
9.2.4	Query Worsening Minimization . . . . .	218
9.2.5	Improving Pseudo-Relevance Feedback (PRF) . . . . .	219
9.2.6	Promises of PageRank in Term Weighting/Source Code Retrieval . . . . .	219
9.2.7	Word Embedding Technology in Query Reformulation/Code Search . . . . .	219
9.2.8	Promises of Stack Overflow in Query Reformulation/Code Search . . . . .	220
9.2.9	Word Embeddings Technology for Bug Understanding/Diagnosis . . . . .	221
9.2.10	Query Reformulation as a Feasible Choice for Improved Bug Localization . . . . .	221
<b>A</b>	<b>Replication Packages</b>	<b>240</b>
A.1	STRICT . . . . .	240

A.2	ACER	240
A.3	BLIZZARD	240
A.4	BLADER	240
A.5	RACK	240
A.6	NLP2API	241
A.7	Other PhD Projects	241
<b>B</b>	<b>BugDoctor</b>	<b>242</b>
B.1	Download	242
B.2	Configuration Setup	242
B.3	Enabling BugDoctor in the IDE	243
B.4	BugDoctor User Interfaces	243
B.5	Loading an Issue Report (e.g., Change Request, Bug Report)	247
B.6	Concept Location with BugDoctor	248
B.7	Bug Localization with BugDoctor	251
B.8	Code Example Search with BugDoctor	252

# LIST OF TABLES

1.1	Thesis Contribution Overview . . . . .	6
3.1	An Example Change Request (Issue #303705, eclipse.jdt.ui) . . . . .	26
3.2	A Working Example of Query Suggestion by STRICT . . . . .	36
3.3	Experimental Dataset . . . . .	37
3.4	Comparison of Query Effectiveness between STRICT and Baseline Queries . . . . .	40
3.5	Effectiveness Details of STRICT Query vs. Baseline Queries, (Title+Description) <sub>code</sub> . . . . .	42
3.6	Document Retrieval Performance of STRICT Queries . . . . .	43
3.7	Retrieval Performance of TextRank, POSRank and WK-Core . . . . .	47
3.8	Comparison between Proposed and Traditional Term Weights . . . . .	47
3.9	Comparison of Baseline Query Improvements between STRICT and Existing Techniques . . . . .	48
3.10	Comparison of Query Effectiveness with Existing Query Reformulation Techniques . . . . .	49
3.11	Comparison with Existing Techniques in Document Retrieval . . . . .	53
4.1	An Example Change Request (Issue #31110, eclipse.jdt.debug) . . . . .	64
4.2	A Working Example (Bug #31110, eclipse.jdt.debug) . . . . .	72
4.3	Experimental Dataset . . . . .	73
4.4	Effectiveness of ACER Query against Baseline Query . . . . .	75
4.5	Effectiveness of ACER Variants against Baseline Queries . . . . .	76
4.6	Comparison of ACER's Retrieval Performance with Baseline Queries . . . . .	77
4.7	Comparison between CodeRank and Traditional Term Weights . . . . .	78
4.8	Impact of Stemming on Query Effectiveness . . . . .	79
4.9	Comparison of Query Effectiveness with Existing Techniques . . . . .	80
5.1	A Noisy Bug Report (Issue #31637, eclipse.jdt.debug) . . . . .	87
5.2	A Poor Bug Report (Issue #187316, eclipse.jdt.ui) . . . . .	88
5.3	Working Examples . . . . .	94
5.4	Experimental Dataset . . . . .	95
5.5	Performance of BLIZZARD in Bug Localization . . . . .	97
5.6	Query Improvement by BLIZZARD over Baseline Queries . . . . .	99
5.7	Comparison with IR-Based Bug Localization Techniques . . . . .	102
5.8	Components behind Existing IR-Based Bug Localization . . . . .	103
5.9	Comparison of Query Effectiveness with Existing Query Reformulation Techniques . . . . .	104
6.1	An Example of Low Quality Bug Report (Issue #192756, ECF) . . . . .	112
6.2	Experimental Dataset (Subject Systems & Bug Reports) . . . . .	122
6.3	Performance of BLADER <sub>BL</sub> in Bug Localization . . . . .	123
6.4	Comparison of Query Effectiveness with Baseline Queries . . . . .	125
6.5	Comparison with Existing Bug Localization Techniques . . . . .	126
6.6	Comparison of Query Effectiveness with Existing Query Reformulation Techniques . . . . .	127
6.7	Comparison with Existing Studies using Feature Matrix . . . . .	128
7.1	API Packages for Exploratory Study . . . . .	138
7.2	Research Questions Answered using Exploratory Study . . . . .	139
7.3	Keywords Intended for Code Search . . . . .	144
7.4	Code Search Keywords Found in Tutorial Sites . . . . .	145
7.5	An Example of Query Reformulation using RACK . . . . .	155
7.6	Research Questions Answered using our Experiment . . . . .	157
7.7	Performance of RACK . . . . .	162
7.8	Role of Proposed Heuristics– KAC, KPAC and KKC . . . . .	164

7.9	Impact of Different Query Term Selection . . . . .	166
7.10	Comparison of API Recommendation Performance with Existing Techniques (for various Top-K Results) . . . . .	170
7.11	Comparison of Source Code Retrieval Performance with Baseline Queries . . . . .	173
7.12	Improvement of Baseline Queries by RACK . . . . .	176
7.13	Comparison of Code Retrieval Performance with Existing Techniques . . . . .	177
7.14	Comparison of Query Improvements with Existing Techniques . . . . .	180
7.15	Comparison with Popular Web/Code Search Engines . . . . .	182
7.16	Comparison among the Traditional Code Search Engines . . . . .	184
8.1	Reformulations of an NL Query for Improved Internet-scale Code Search . . . . .	193
8.2	Performance of NLP2API in Relevant API Suggestion . . . . .	202
8.3	Comparison with the State-of-the-art in API Class Suggestion . . . . .	205
8.4	Impact of Reformulations on Generic NL Queries . . . . .	206
8.5	Comparison of Query Effectiveness with Existing Query Reformulation Techniques . . . . .	207
8.6	Comparison with Popular Web/Code Search Engines . . . . .	209

# LIST OF FIGURES

1.1	(a) An example of noisy bug report (noisy query) and (b) Reformulated search query suggested by BugDoctor. The noisy query returns the buggy code at the 53 <sup>rd</sup> position whereas the reformulated query returns that at the 1 <sup>st</sup> position within the result list. . . . .	3
1.2	(a) An example of software change request (query) and (b) Reformulated search query suggested by BugDoctor. The given query returns the code of interest at the 14 <sup>th</sup> position whereas the reformulated query returns that at the 1 <sup>st</sup> position within the result list. . . . .	4
1.3	(a) An example of poor bug report (poor query) and (b) Reformulated search query suggested by BugDoctor. The poor query returns the buggy code at the 12 <sup>th</sup> position whereas the reformulated query returns that at the 3 <sup>rd</sup> position within the result list . . . . .	4
2.1	Automatic query reformulations in local code search . . . . .	15
2.2	Automatic query reformulations in the Internet-scale code search . . . . .	17
3.1	Text graphs of the change request in Table 3.1 – (a) using word co-occurrences, and (b) using syntactic dependencies . . . . .	25
3.2	Schematic diagram of the proposed query reformulation technique–STRICT . . . . .	27
3.3	Improvement, worsening and preserving of the baseline queries by our proposed technique – STRICT . . . . .	41
3.4	Comparison of the document retrieval performance of STRICT queries against baseline queries in terms of (a) Hit@10, (b) MAP@10, and (c) MRR@10 . . . . .	43
3.5	Comparison of STRICT queries with baseline queries for Top 1 to 100 results in terms of (a) MAP@K and (b) Hit@K . . . . .	45
3.6	Role of three term weighting algorithms in the improvement, worsening and preserving of the baseline queries . . . . .	50
3.7	Impact of the adopted parameters and thresholds – (a,b) suggested query length, (c) use of data re-sampling, and (d) use of machine learning algorithm . . . . .	51
3.8	Comparison of baseline query improvements or worsening between our technique, STRICT, and the existing techniques . . . . .	52
3.9	Comparison between queries of STRICT and the queries of existing approaches in terms of their (a) Hit@10, (b) MAP@10, and (c) MRR@10 . . . . .	53
3.10	Comparison between queries of STRICT and queries from the existing approaches in terms of (a) MAPK and (b) Hit@K . . . . .	54
3.11	Stage I - Distribution of the grades for study tasks . . . . .	54
3.12	Stage II - User evaluation of the proposed prototype in terms of <b>EI</b> =Ease of Installation, <b>DQ</b> =Documentation Quality, <b>UF</b> =Usefulness of Features, <b>LF</b> =Likelihood of Features, <b>QSQ</b> =Quality of Suggested Queries, <b>MER</b> =Manual Effort Reduction, <b>TSP</b> =Time Saving Potential . . . . .	55
4.1	An example term graph generated by CodeRank for the source code of Fig. 4.2 . . . . .	63
4.2	Source code used for automated query reformulation . . . . .	65
4.3	Schematic diagram of the proposed query reformulation technique–ACER . . . . .	65
4.4	Comparison of query improvement between CodeRank and traditional term weights for (a) Top K=1 to 10 and (b) Top K=1 to 30 reformulated query terms . . . . .	78
4.5	Improved queries by reformulation from method signatures and field signatures using (a) CodeRank (CR) and (b) Term Frequency (TF). (c) ACER vs. TF (all content) . . . . .	79
4.6	Effectiveness of ACER queries for (a) Top-10 and (b) Top-30 reformulated terms . . . . .	81
4.7	Comparison of (a) query effectiveness, and (b) retrieval performance . . . . .	81
5.1	Schematic diagram of the proposed query reformulation technique –BLIZZARD–(A) Bug report classification and (B) Search query suggestion . . . . .	88
5.2	Trace graph of stack traces in Table 5.1 . . . . .	90

5.3	Comparison of BLIZZARD with baseline technique in terms of (a) MAP@K and (b) MRR@K	98
5.4	Impact of query reformulation length on the MAP@10 of our technique–BLIZZARD	98
5.5	Quality improvement of (a) noisy and (b) poor baseline queries by our technique–BLIZZARD	99
5.6	Comparison of (a) MAP@K and (b) Hit@K with the state-of-the-art IR-based bug localization techniques	105
5.7	Comparison of Hit@10 across all subject systems	105
5.8	Comparison of (a) MRR@10 and (b) MAP@10 with existing techniques across the subject systems	106
6.1	Schematic diagram of the proposed query reformulation technique –BLADER– (A) Construction of a semantic hyperspace and (B) Reformulation of a query for bug localization	113
6.2	Comparison of our approach, BLADER <sub>BL</sub> , with the baseline approach in bug localization using (a) MAP and (b) MRR	122
6.3	Impact of our adopted thresholds, parameters and choices – (a) Multiple reformulation candidates, (b) Number of candidate source terms, (c) Machine learning algorithm for the best query selection, and (d) Corpus for learning word embeddings	124
6.4	Comparison of our approach with the existing techniques in bug localization using (a) MAP, and (b) MRR for top 1 to 10 results	128
6.5	(a) Overlap of the successfully localized bugs between BLADER <sub>BL</sub> and the state-of-the-art, and (b) Overlap of the improved queries between BLADER <sub>QR</sub> and the state-of-the-art approaches	129
6.6	Comparison of our approach with the existing techniques in query reformulation using very low quality queries	130
7.1	An example of (a) Stack Overflow question and (b) its accepted answer	136
7.2	Frequency distribution for core API classes – (a) API frequency PMF, (b) API frequency CDF	140
7.3	Frequency distribution for core and non-core API classes over the extended dataset – (a) API frequency PMF, (b) API frequency CDF	140
7.4	Frequency distribution of unique API classes from core packages – (a) Distinct API frequency PMF, (b) Distinct API frequency CDF	141
7.5	Frequency distribution of unique API classes from core and non-core packages – (a) Distinct API frequency PMF, (b) Distinct API frequency CDF	141
7.6	Coverage of API classes from core packages by Stack Overflow answers	143
7.7	Coverage of API classes from (a) core and (b) non-core packages by Stack Overflow answers (extended dataset)	144
7.8	Use of core API packages in the Stack Overflow answers	145
7.9	Use of (a) core and (b) non-core API packages in the Stack Overflow answers (extended dataset)	146
7.10	Coverage of keywords from the collected queries in Stack Overflow questions	147
7.11	Collected search query keywords in Stack Overflow– (a) Keyword frequency PMF (b) Keyword frequency CDF	147
7.12	Schematic diagram of the proposed query reformulation technique –RACK–(a) Construction of token-API mapping database, (b) Translation of a code search query into relevant API classes	148
7.13	Hit@K, Mean Average Precision@K, and Mean Recall@K of RACK using (a) non-weighted version (i.e., dashed line) and (b) weighted version (i.e., solid line)	163
7.14	(a) Hit@K of RACK, (b) Mean Average Precision@K (MAP@K) of RACK, and (c) Mean Recall@K (MR@K) of RACK for three heuristics–KAC, KPAC and KKC	165
7.15	(a) Mean Average Precision@10 (MAP@10), and (b) Mean Recall@10 (MR@10) of RACK for different values of the heuristic weights– $\alpha$ and $\beta$	168
7.16	Performance of RACK for different $\delta$ thresholds with (a) Top-5 results and (b) Top-10 results considered	168
7.17	Performance of RACK for different $\gamma$ thresholds with (a) Top-5 results and (b) Top-10 results considered	169
7.18	Comparison of API recommendation performances with the existing techniques–(a) Hit@K, (b) Mean Reciprocal Rank@K, (c) Mean Average Precision@K, and (d) Mean Recall@K	171
7.19	Comparison of API recommendation with existing techniques using box plots	172

7.20	Comparison of code retrieval performance with the baseline queries in terms of (a) Hit@K and (b) MRR@K . . . . .	175
7.21	Comparison of QE distribution with baseline queries across (a) 4K-Corpus, (b) 256K-Corpus and (c) 769K-Corpus . . . . .	175
7.22	Comparison of code retrieval performance with existing techniques using (a,b) 4K-Corpus, (c,d) 256K-Corpus and (e,f) 756K-Corpus . . . . .	179
7.23	Comparison of QE distribution with the state-of-the-art using (a) 4K-Corpus, (b) 256K-Corpus, and (c) 769K-Corpus . . . . .	181
7.24	Comparison of RACK with popular web/code search engines . . . . .	183
8.1	An example code snippet for the programming task– “Convert image to grayscale without losing transparency” – (taken from [9]) . . . . .	193
8.2	Schematic diagram of the proposed query reformulation technique–NLP2API . . . . .	194
8.3	API co-occurrence graph for code segment in Fig. 8.1 . . . . .	196
8.4	Performance of NLP2API in API class suggestion for various Top-K results . . . . .	202
8.5	Impact of (a) PRF size (M), and (b) Candidate API list size (N) on relevant API class suggestion from Stack Overflow . . . . .	203
8.6	Comparison between Borda count and Query-API proximity in estimating API relevance using (a) accuracy, (b) reciprocal rank, (c) precision, and (d) recall . . . . .	204
8.7	Reformulated vs. baseline query using (a) Top-10 accuracy and (b) MRR@10 . . . . .	205
8.8	Comparison between popular web/code search engines and NLP2API in relevant code segment retrieval using (a) MAP@K and (b) NDCG@K . . . . .	209
B.1	Setting up custom configurations for BugDoctor . . . . .	242
B.2	Enabling BugDoctor with (a) main menu option and (b) context menu option . . . . .	243
B.3	BugDoctor Dashboard: (a) Query execution panel, (b) Bug report panel, (c) Query reformulation panel, and (d) Code search results panel . . . . .	244
B.4	BugDoctor Utility Dashboard: (a) API suggestion & query execution panel, (b) Query expansion panel, and (c) Code viewer . . . . .	245
B.5	Code Example Dashboard: (a) Top-K relevant code examples, and (b) Code viewer . . . . .	246
B.6	Loading of an issue report: (1) Click the button, (2) Choose the report, and (3) View the report within the IDE . . . . .	247
B.7	Concept location with query reduction: (1-2) Open a change request, i.e., given query, (3-4) Keyword suggestion, (5-6) Reduced query, (7) Code search, and (8) Located concept within the Top-10 results . . . . .	248
B.8	Concept location with baseline query: (1-3) Selection of report title as a baseline query, (4) Code search, and (5) Concept not located within the Top-10 results . . . . .	249
B.9	Concept location with query expansion : (1) Selection of report title as a given query, (2-3) Query expansion, (4-5) Expanded query, (6) Code search, and (7) Concept located within the Top-10 results . . . . .	250
B.10	Bug localization with query reduction: (1-2) Open a bug report, i.e., given query, (3-4) Keyword suggestion, (5-6) Reduced query, (7) Code search, and (8) Localized buggy class as the topmost result, and (9) Analysis for bug fixing . . . . .	251
B.11	Code example search with query expansion: (1) Given programming task, i.e., given query, (2-3) Relevant API suggestion, (4-5) Expanded query, (6) Code example search, (7) Retrieved code example, (8) Original code location on the web, and (9) Click the button for Top-K code examples . . . . .	252
B.12	Relevant code examples: (1) Top-K code examples, and (2) Code example viewer . . . . .	253

## LIST OF ABBREVIATIONS

ACR	API Class Rank
API	Application Programming Interface
BL	Bug Localization
BR	Bug Report
CART	Classification and Regression Tree
CBOW	Continuous Bag of Words
CDF	Cumulative Density Function
CT	Candidate Token
ECF	Eclipse Communication Framework
HS	Hopkins Statistic
HTML	Hyper Text Markup Language
IDCG	Ideal Discounted Cumulative Gain
IDE	Integrated Development Environment
IRC	Internet Relay Chat
IR	Information Retrieval
JDK	Java Development Kit
JDT	Java Development Tools
KAC	Keyword-API Co-occurrence
KKC	Keyword-Keyword Coherence
KPAC	Keyword Pair-API Co-occurrence
MAP	Mean Average Precision
MCAS	Maneuvering Characteristics Augmentation System
MRD	Mean Rank Difference
MRR	Mean Reciprocal Rank
MWW	Mann-Whitney Wilcoxon
NDCG	Normalized Discounted Cumulative Gain
NL	Natural Language
PA	Polygon Area
PE	Program Entity
PMF	Probability Mass Function
POSR	POSRank
POS	Parts of Speech
PRF	Pseudo-Relevance Feedback
Q&A	Question & Answering
QE	Query Effectiveness
QR	Query Reformulation
RC	Reformulation Candidate
RF	Relevance Feedback
RQ	Research Question
RSV	Robertson Selection Value
SAN	Spreading Activation Network
SCP	Spatial Code Proximity
SE	Software Engineering
ST	Stack Trace
TF-IDF	Term Frequency $\times$ Inverse Document Frequency
TR	TextRank
TW	Term Weight
UI	User Interface
VSM	Vector Space Model
WE	Word Embedding
WSR	Wilcoxon Signed Rank



# CHAPTER 1

## INTRODUCTION

### 1.1 Motivation

Software bugs and failures cost trillions of dollars every year. In 2017 alone, 606 software bugs<sup>1</sup> cost around \$1.7 trillion with 3.7 billion people affected and 314 companies impacted. During 2009–2018, about half a million elderly British women missed their mammography tests due to scheduling errors caused by a software bug<sup>2</sup>, which might have led to hundreds of premature deaths. Back in 1985-1987, four Canadian patients also lost their lives and two were heavily injured due to fatal software bugs in Therac-25 radiation therapy system<sup>3</sup>. All these unfortunate tragedies demonstrate the serious consequences of software bugs and failures. There have been active researches for the last five decades to prevent software bugs, errors and failures. However, given their high costs and deadly consequences, further researches are warranted more than ever.

Software developers attempt to solve hundreds of bugs and failures every day. For example, Mozilla Corporation receives  $\approx 300$  software bug reports every day that need urgent fixes [38, 249]. *Bug report* is a text document that explains the encountered errors or failures in a software system. For example, Fig. 1.1-(a) shows an example bug report that explains an error encountered in the Eclipse IDE. While software bugs and errors are already hard to tackle, developers also receive hundreds if not thousands of software change requests during maintenance [88]. *Change request* is a text document that warrants for either new software features or enhancements to the existing features in a software system. Fig. 1.2-(a) shows an example change request that warrants for enhancement in the custom search feature of Eclipse IDE. Resolving the software bugs and addressing the change requests are two major parts of *software maintenance*. Finding and fixing the bugs consume about 50% of the development time and efforts which amount to 20% of the total maintenance costs [4, 28, 81]. On the other hand, adding new features to the existing software systems claims even up to 60% of the total maintenance costs [88].

The very first challenge of any software maintenance is to identify the exact locations in the software code that need to be repaired or modified. One needs to find out the exact locations where the bug should be fixed or the existing feature that should be enhanced. Unfortunately, given million lines of code and inherent complexities in the modern software systems, identification of such locations is extremely challenging.

---

<sup>1</sup><https://tek.io/2FBNl2i>

<sup>2</sup><https://bit.ly/2E1fYap>

<sup>3</sup><https://bit.ly/2KU9IR2>

Locating the buggy code against a bug report is called **bug localization** [276]. On the other hand, locating the target code against a change request is known as **concept location** [98, 120]. Thus, both bug localization and concept location are a special type of code search that is performed within a software codebase. Besides these specialized searches, developers also search for relevant, reusable code on the web (e.g., GitHub [31]) to implement various programming tasks as a part of software maintenance. This search is known as **Internet-scale code search** in the literature [45, 151].

Every search operation requires a query that reflects the information needs. During maintenance, software developers often (1) choose a few important keywords [120] or (2) use the whole texts [220] from an *issue report* (i.e., change request or bug report) as a search query. During code search on the web, they also choose a few keywords as an ad hoc search query [45]. Then the query is executed with a search engine to find out (1) the exact code locations (within a software system) that need to be repaired or enhanced, and (2) the relevant code examples (from thousands of online projects) that can be reused. Unfortunately, even the experienced developers often fail to choose the right search queries [83, 120, 125, 142]. Multiple developer studies [120, 125] report that the search queries chosen by developers could fail up to **88%** of the time in localizing the desired code (e.g., bugs, concepts, reusable code). That is, whether it is bug localization, concept location or Internet-scale code search, appropriate query construction is a major challenge. Thus, software developers are badly in need of automated supports for query construction during the code search.

## 1.2 Problem Statement

Software bugs are pervasive and code changes are inevitable in modern software systems [88]. Whether it is a bug or a programming concept, they need to be localized correctly using code search. Appropriate query construction is a major challenge in any type of code search, let it be bug localization, concept location or even general-purpose, Internet-scale code search. There have been active researches [64, 65, 84, 95, 98, 104, 109, 120, 134, 135, 144, 151, 168, 226, 231, 251, 265, 274] on automated query construction for bug localization, concept location and general-purpose code search. These studies accept either an issue report or a set of generic keywords as a query, and then deliver an improved version of the query through *query reformulations*. Such reformulations involve removal of noisy keywords, addition of complementary keywords, and replacement of the poor keywords with more appropriate ones. Although there have been a substantial body of works, the existing literature on automated query reformulation is far from adequate. According to our systematic literature review [17], they suffer from four major limitations as follows:

**(a) Term dependency overlooked:** Determining relative importance of the candidate keywords is a primary step of automated query reformulations. Existing studies [49, 96, 98, 120, 139, 158, 273] extensively use TF-IDF [114] in term weighting and then choosing the highly weighted keywords from change requests or bug reports as queries. Unfortunately, TF-IDF suffers from a major limitation [53, 153]. It fails to capture the semantic dependencies between a given word and its surrounding words. The *semantics* of a word are often

### Bug 31637 - should be able to cast "null"

When trying to debug an application the variables tab is empty.  
Also when I try to inspect or display a variable I get the following  
error logged in the eclipse log file:

```
!ENTRY org.eclipse.jdt.debug 4 120 Feb 12, 2003 11:11:29.503
!MESSAGE Internal error logged from JDI Debug:
!STACK 0
java.lang.NullPointerException
at org.eclipse.jdt.internal.debug.core.model.JDIValue.toString(JDIValue.java:362)
at org.eclipse.jdt.internal.debug.eval.ast.instructions.Cast.execute(Cast.java:88)
at org.eclipse.jdt.internal.debug.eval.ast.engine.Interpreter.execute(Interpreter.java:44)
at org.eclipse.jdt.internal.debug.eval.ast.engine.EvaluationThread$1$EvaluationRunnable.
run(EvaluationThread.java:158)
at org.eclipse.jdt.internal.debug.core.model.JDIThread.runEvaluation(JDIThread.java:600)
at org.eclipse.jdt.internal.debug.eval.ast.engine.EvaluationThread.
doEvaluation(EvaluationThread.java:180)
at org.eclipse.jdt.internal.debug.eval.ast.engine.EvaluationThread.
access$2(EvaluationThread.java:142)
at org.eclipse.jdt.internal.debug.eval.ast.engine.EvaluationThread$1.
run(EvaluationThread.java:116)
at java.lang.Thread.run(Thread.java:536)
I am using Eclipse 2.1 (M5 / RCO).
```

53

(a)

Bug should be able to cast null + NullPointerException +  
toString JDIValue EvaluationThread EvaluationRunnable String  
access Evaluation run execute Thread runEvaluation

1

(b)

**Figure 1.1:** (a) An example of noisy bug report (noisy query) and (b) Reformulated search query suggested by BugDoctor. The noisy query returns the buggy code at the 53<sup>rd</sup> position whereas the reformulated query returns that at the 1<sup>st</sup> position within the result list.

determined by its surrounding words within a particular context (e.g., sentence) [157, 272]. For example, the word "bank" has multiple meanings. It could mean either "the land alongside a river" or "a financial institution", which is determined by its surrounding words. However, TF-IDF overlooks such a crucial aspect (i.e., term dependency) during determining the term importance. Thus, the existing studies might produce such queries that fail to localize the buggy code or the code implementing a concept of interest.

**(b) Sole reliance on source code vocabulary:** Existing studies [84, 96, 98, 251] often reformulate a poor query with important keywords taken from the source code where TF-IDF is used for keyword selection. However, TF-IDF was originally targeted for regular texts (e.g., news articles) rather than source code. Regular texts and source code differ significantly from each other in their syntax, semantics and structures [102, 233]. While regular texts are rich in vocabulary, source code is poor in vocabulary but rich in structures [102]. Unfortunately, TF-IDF fails to leverage such structural aspect of the code and solely relies on the vocabulary aspect during keyword selection. Thus, existing studies might fail to improve the poor search queries or even worse, deliver poorer queries due to reformulation with inappropriate keywords.

**(c) Overlooking the quality aspects of bug reports:** Bug reports often contain a mix of unstructured regular texts and structured program elements (e.g., class names). These structured elements often provide useful hints about the location of an encountered bug. However, a significant fraction of the reports ( $\approx 30\%$ )

### Bug 303705 - [search] Custom search results not shown hierarchically in the java search results view

Consider an instance of `org.eclipse.search.ui.text.Match` with an element that is neither an `IResource` nor an `IJavaElement`. It might be an element in a class diagram, for example. (14)

When such an element is reported, it will be shown as a plain, flat element in the otherwise hierarchical java search results view. This is because the `LevelTreeContentProvider` and its superclasses only check for `IJavaElement` and `IResource`. (a)

I propose to also check for `IAdaptable` and try to adapt to `IJavaElement` and `IResource`, if the other checks fail.

element IResource Provider Level Tree (1) (b)

**Figure 1.2:** (a) An example of software change request (query) and (b) Reformulated search query suggested by BugDoctor. The given query returns the code of interest at the 14<sup>th</sup> position whereas the reformulated query returns that at the 1<sup>st</sup> position within the result list.

### Bug 192756 - [IRC] On channel join, get rid of 'entered' spam in

If you join a big channel, you get a ton of "xxx entered". I think on channel entry, we don't show these messages. We should show these messages in maybe the "server tab", i.e., `irc.freenode.net`, similar to how other IRC clients do it. (12) (a)

{title} + {description} + connect invitation handle message room chat user send (3) (b)

**Figure 1.3:** (a) An example of poor bug report (poor query) and (b) Reformulated search query suggested by BugDoctor. The poor query returns the buggy code at the 12<sup>th</sup> position whereas the reformulated query returns that at the 3<sup>rd</sup> position within the result list

could be *poor* containing no localization hints. On the contrary, about 15% of the reports are *noisy*, which are crowded with too much structured information (e.g., stack traces). Both these bug reports verbatim do not make good search queries for bug localization [193, 248]. However, majority of the existing studies [130, 167, 207, 220, 230, 249, 276] use almost verbatim texts from a bug report as a search query for the bug localization. Thus, their queries could be either *noisy* due to stack traces or *poor* due to the lack of localization hints. As a result, their queries might fail to localize the reported software bugs.

**(d) Relevant API selection impaired:** As a frequent practice, developers often issue free-form natural language queries for searching relevant code snippets on the web (e.g., GitHub). Unfortunately, these queries hardly lead to any relevant results (e.g., only 12%) [45]. Several existing studies [63, 147, 152, 243, 271] attempt to reformulate a free-form query with relevant API classes. The baseline idea is to reduce the lexical gap between the query and potentially relevant code snippets. However, they simply rely on the *lexical similarity* between a given query and the candidate API classes or corresponding API documentations for relevant API selection. Thus, the existing approaches might fail to reformulate the query if it is not lexically

similar to the relevant APIs and their official documentations. As a result, their queries might not be able to retrieve the relevant code snippets from the web.

## 1.3 Our Contribution

Any changes to existing software systems to resolve or prevent these bugs or failures also cost billions of dollars every year [1, 28]. Thus, identifying the exact locations in the source code that need to be repaired is extremely important. Post-release changes to the software features also claim up to 60% of the total maintenance costs. Thus, locating the software code that needs to be either enhanced or reused is also equally important. For these search tasks, various traditional code search methods – *bug localization*, *concept location* and *Internet-scale code search* – are used which are frequently impaired by poor or noisy search queries. In this thesis, we tackle the challenges of poor and noisy queries, and significantly advance the current state of query reformulation research. In particular, we conduct six different studies (Table 1.1) where first and second studies improve concept location, third and fourth studies improve bug localization, and finally fifth and sixth studies improve general-purpose, Internet-scale code search, by incorporating automatically reformulated search queries into these tasks. Finally, we combine all six approaches above, and develop a novel tool namely **BugDoctor**, an Eclipse IDE plug-in (Appendix B). It assists the developers in concept location, bug localization, and Internet-net scale code search with reformulated queries so that they can localize their code of interest (e.g., software bugs, programming concepts, reusable code examples) with less effort and less development time spent. We briefly introduce each of our studies as follows.

### 1.3.1 Concept Location

Two of our studies support concept location task with automated query reformulations.

(a) **STRICT**: We design a novel query reformulation approach (Chapter 3) that accepts a change request as an initial query, identifies appropriate query keywords from the request texts using multiple graph-based term weighting algorithms (e.g., TextRank [153], POSRank [53, 153]), and then delivers an improved, reformulated search query for concept location. Fig. 1.2-(a) shows an example change request (query) whereas Fig. 1.2-(b) shows the corresponding reformulated query delivered by this approach.

(b) **ACER**: We design another novel query reformulation approach (Chapter 4) that accepts a poor query, identifies complementary keywords from the relevant source code using a graph-based term weighting algorithm (CodeRank [189]), and then delivers an improved query (poor query + complementary keywords) (e.g., Table 4.1) for concept location.

### 1.3.2 Bug Localization

Two of our studies support bug localization task with automated query reformulations.

**Table 1.1:** Thesis Contribution Overview

Approach	Working Context	Input	Output	Novel Contribution
S1: STRICT	concept location	change request	reformulated query	graph-based term weighting on change request
S2: ACER	concept location	developer query	reformulated query	graph-based term weighting on source code
S3: BLIZZARD	bug localization	bug report	reformulated query	report quality dynamics in search query improvement
S4: BLADER	bug localization	bug report	reformulated query	clustering tendency analysis for search query improvement
S5: RACK	Internet-scale code search	programming task description	reformulated query	crowd knowledge in query-API relevance estimation
S6: NLP2API	Internet-scale code search	programming task description	reformulated query	query-API semantic distance calculation for query reformulation

(c) **BLIZZARD:** We design a novel query reformulation approach (Chapter 5) that accepts either a noisy bug report (e.g., Fig. 1.1-(a)) or a poor bug report (e.g., Fig. 1.3-(a)) as a query, identifies appropriate keywords by leveraging the structured entities and reporting quality dynamics of the report, and then delivers an improved, reformulated search query (e.g., 1.1-(b), 1.3-(b)) for the bug localization.

(d) **BLADER:** We design another novel query reformulation approach (Chapter 6) that accepts a poor bug report (e.g., Fig. 1.3) as a query, identifies complementary keywords from the relevant source code by analysing clustering tendency between the query and the candidate keywords, and then delivers an improved, reformulated version (poor query + complementary keywords) of the given poor query (e.g., Table 6.1) for the bug localization.

### 1.3.3 Internet-scale Code Search

Two of our studies support Internet-scale code search with automated query reformulations.

(e) **RACK:** We design a novel query reformulation approach (Chapter 7) that accepts a generic natural language query on a programming task, identifies relevant API classes for the task by harnessing crowd generated knowledge at Stack Overflow, and then delivers an improved, reformulated query (generic query + relevant API classes) (e.g., Table 7.5) for Internet-scale code search. Query reformulation using the relevant API classes reduces the lexical gap between the query and the solution code.

(f) **NLP2API:** We design another novel query reformulation approach (Chapter 8) that accepts a programming task description as a query, identifies relevant API classes by mining Stack Overflow Q&A threads and by determining semantic distance between the query and the candidate API classes, and then delivers an improved, reformulated version (generic query + relevant API classes) of the given query for general-purpose, Internet-scale code search.

Each of these studies has been evaluated extensively using appropriate dataset such as actual bug reports, change requests and changed source code documents. They are also compared against multiple baselines including the state-of-the-art to demonstrate their superiority. Experimental findings suggest that our approaches not only improve the given search queries significantly but also outperform the state-of-the-art with statistically significant margins. Furthermore, they support the developers in concept location, bug localization, and Internet-net scale code search so that they can locate their concepts, software bugs and relevant code examples respectively with less efforts and time spent. Thus, from a technical point of view, this PhD dissertation makes five major contributions to the existing body of knowledge as follows.

**(a) Introducing a novel, effective algorithm for search keyword selection in Software Engineering tasks:** TF-IDF [114] (1) overlooks the semantic or syntactic dependencies among the words within a particular context (e.g., sentence), and also (2) fails to leverage the structural aspect of a body of texts (e.g., bug report, source code document), which are crucial to term importance [53, 153]. Thus, it might not be sufficient enough for determining term importance and for selecting appropriate search keywords. Our works [187, 189, 191, 192] (Chapters 3, 4, 5) propose and design a *novel* algorithm that not only captures the dependencies among the candidate keywords but also leverages the structural aspect of a document (containing the keywords) in selecting the appropriate keywords. Furthermore, our algorithm outperforms TF-IDF in selecting search keywords from bug reports [192], change requests [191], source code documents [189] and even from the noisy stack traces [192]. Thus, we offer a better solution for search keyword selection in the context of Software Engineering.

**(b) Incorporation of bug report quality dynamics in query reformulations for improved bug localization:** Earlier studies [220, 250, 255, 276] overlook the quality aspect of bug reports, and use almost verbatim texts from noisy (e.g., Fig. 1.1-(a)) and poor bug reports (e.g., Fig. 1.3-(a)) as queries for bug localization. Thus, they potentially use noisy or poor queries which often fail to localize the buggy entities [248]. My PhD works [192, 193] analyse the quality of a bug report (a.k.a., given query) and apply appropriate reformulations to the query based on its quality unlike the earlier approaches. In particular, we (1) refine the noisy bug report by discarding the noisy keywords, (2) complement the poor bug report with appropriate keywords collected from the relevant source code, and (3) then deliver an improved, reformulated query (e.g., Fig. 1.1-(b), 1.3-(b)) for bug localization. Such opportunistic reformulations significantly improve the given queries in bug localization [192].

**(c) Exploiting structures from source code documents in search query improvement:** Existing studies [98, 150, 213] overlook the structural aspect of source code documents and treat them as regular text documents during keyword selection from them. My PhD works [188, 189, 192] (Chapters 4, 5) leverage the query contexts and structures from the source code documents in reformulating a given query. In particular, we treat each source code document as a network of various structural constructs (e.g., method signatures, field signatures) rather than a *bag of words*. We leverage these structures, and prepare multiple reformulation candidates using graph-based term weighting for a given search query. Then we deliver the best candidate

as a reformulated query using machine learning. Such a novel use of structural constructs from source code documents has significantly improved a given query in concept location [189].

**(d) Exploiting crowd generated knowledge in search query improvement:** Earlier studies (1) mine open source projects [109, 265, 266, 274], web search logs [181, 223] or official API documentations [147, 243, 271] and (2) use English language thesaurus (e.g., WordNet [134, 144]) to reformulate a given query with synonyms, similar words and API classes. Stack Overflow has been a popular Q&A site for programming issues or API usage examples with 14 million questions, 22 million answers and 10 million registered users [58]. However, its potential for query improvement with API related resources was largely unexplored. My PhD works [71, 194, 195, 201, 204, 206] (Chapters 6, 7, 8) demonstrate the high potential of Stack Overflow for improving search queries intended for bug localization and for Internet-scale code search. In particular, we complement a generic query on a programming task with relevant API classes where the relevance between the query and the API classes is learned from the large technical crowd of Stack Overflow. The knowledge on API relevance is non-trivial and can only be gained through actual work experience. Thus, our query reformulation approach that exploits such invaluable crowd knowledge is a novel addition to the literature. Furthermore, the extension using relevant API classes has also significantly improved the generic queries in general-purpose, Internet-scale code search [194, 206].

**(e) Exploiting word embeddings and clustering tendency in search query improvement:** Several existing studies [71, 194, 268, 274] use word embeddings technology, calculate *semantic distance* between a given query and a candidate API class, and then expand the query with semantically close API classes. While the semantic distance idea might work for Internet-scale code search, it might not be sufficient enough for the specialized code search such as bug localization. The latter search warrants more precision since the changes in wrong code locations could be very costly. My PhD work [182] (Chapter 6) goes beyond semantic distance, and exploits *clustering tendency* in improving the search queries for bug localization. In particular, we construct two reformulation candidates for a given poor query, and determine the clustering tendency between the query and the reformulation candidates using advanced metrics from computational geometry (e.g., Hopkins statistics [108]) and word embeddings technology [54, 155]. Then, we deliver the best candidate as a reformulated query using machine learning. Our approach significantly improves the poor queries in terms of their bug localization performance. Such a solution is a novel addition to the existing body of knowledge on automated query reformulation targeting any types of code search.

Besides the above thesis works, we have conducted significant researches on code review automation [190, 200, 202, 203], mining software repositories [160, 161, 186, 196] and source code re-documentation [199] during my PhD study. My MSc works [183, 184, 185, 197, 198] also develop tools for web/code searches to assist the developers in solving programming errors and exceptions encountered within their IDE.



## 1.4 Related Publications

Five out of six studies from this thesis are published in different premier conferences and journals. We provide a list of publications that were produced from this PhD study (2014–2019). Due to strong relevance, six papers from my MSc study (2012–2014) are also included here. In majority of these papers, I am the primary author, and the studies were conducted by me under the supervision of Dr. Chanchal K. Roy. While I conducted the experiments and wrote the papers, the co-authors took part in advising, editing, and reviewing the papers. I also co-supervised three published works where I am the second/third author. In these works, I co-supervised the whole life cycle of each project from brainstorming the research ideas to co-authoring the papers.

- (1) **M. Masudur Rahman**, “*Supporting Code Search with Context-Aware, Analytics-Driven, Effective Query Reformulation*”, In Proceeding of The 41st ACM/IEEE International Conference on Software Engineering (Companion volume, Doctoral Symposium Track) (**ICSE 2019**), pp. 226–229, Montreal, Canada, May, 2019 (Acceptance rate: **9/31=29.00%**)
- (2) S. Mondal, **M. Masudur Rahman** and C. K. Roy, “*Can Issues Reported at Stack Overflow Questions be Reproduced? An Exploratory Study*”, In Proceeding of The 16th International Conference on Mining Software Repositories (**MSR 2019**), pp. 479–489, Montreal, Canada, May, 2019 (Acceptance rate: **32/126=25.40%**)
- (3) Rodrigo F. G. Da Silva, C. K. Roy, **M. Masudur Rahman**, K. Schneider, K. Paixão and M. Maia, “*Recommending Comprehensive Solutions for Programming Tasks by Mining Crowd Knowledge*”, In Proceeding of The 27th IEEE/ACM International Conference on Program Comprehension (**ICPC 2019**), pp. 358–368, Montreal, Canada, May, 2019 (Acceptance rate: **28/93=30.10%**) (**Featured in Stack Overflow Blog\***)
- (4) **M. Masudur Rahman**, C. K. Roy and David Lo, “*Automatic Query Reformulation for Code Search using Crowdsourced Knowledge*”, Empirical Software Engineering Journal (**EMSE**), 56 pp., (Impact Factor=**4.46**)
- (5) **M. Masudur Rahman** and C. K. Roy, “*Improving IR-Based Bug Localization with Context-Aware Query Reformulation*”, In Proceeding of The 26th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (**ESEC/FSE 2018**), pp. 621–632, Florida, USA, November, 2018 (Acceptance rate: **55/295=19.00%**) (ACM Artifact Badges by peer reviews: **Functional\* + Available\* + Reusable\***)
- (6) **M. Masudur Rahman** and C. K. Roy, “*Effective Reformulation of Query for Code Search using Crowdsourced Knowledge and Extra-Large Data Analytics*”, In Proceeding of The 34th International Conference on Software Maintenance and Evolution (**ICSME 2018**), pp. 516–527, Madrid, Spain,

September, 2018 (Acceptance rate: **37/174=21.00%**) (**TCSE Distinguished Paper Award Nomination\***)

- (7) **M. Masudur Rahman** and C. K. Roy, "*Poster: Improving Bug Localization with Report Quality Dynamics and Query Reformulation*", In Proceeding of The 40<sup>th</sup> International Conference on Software Engineering (**ICSE 2018**), pp. 348–349, Gothenburg, Sweden, May, 2018
- (8) **M. Masudur Rahman** and C. K. Roy, "*NLP2API: Query Reformulation for Code Search using Crowdsourced Knowledge and Extra-Large Data Analytics*", In Proceeding of The 34th International Conference on Software Maintenance and Evolution (Artifact Track) (**ICSME 2018**), pp. 714, Madrid, Spain, September, 2018 (Artifact **Verified and Accepted\***)
- (9) **M. Masudur Rahman** and C. K. Roy, "*Improved Query Reformulation for Concept Location using CodeRank and Document Structures*", In Proceeding of The 32<sup>nd</sup> IEEE/ACM International Conference on Automated Software Engineering (**ASE 2017**), pp. 428-439, Urbana-Champaign, Illinois, USA, October, 2017 (Acceptance rate: **65/314=21.00%**)
- (10) **M. Masudur Rahman** and C. K. Roy and R. G. Kula, "*Predicting Usefulness of Code Review Comments using Textual Features and Developer Experience*", In Proceeding of The 14<sup>th</sup> International Conference on Mining Software Repositories (**MSR 2017**), pp. 215–226, Buenos Aires, Argentina, May, 2017 (Acceptance rate: **37/121=30.60%**)
- (11) **M. Masudur Rahman** and C. K. Roy and David Lo, "*RACK: Code Search in the IDE using Crowdsourced Knowledge*", In Proceeding of The 39<sup>th</sup> International Conference on Software Engineering (Companion Volume) (**ICSE 2017**), pp. 51–54, Buenos Aires, Argentina, May, 2017 (Acceptance rate: **18/57=31.58%**)
- (12) **M. Masudur Rahman** and C. K. Roy, "*STRICT: Information Retrieval Based Search Term Identification for Concept Location*", In Proceeding of The 24<sup>th</sup> IEEE International Conference on Software Analysis, Evolution, and Reengineering (SANER 2017), pp. 79–90, Klagenfurt, Austria, February 2017 (Acceptance rate: **34/140=24.00%**)
- (13) **M. Masudur Rahman** and C. K. Roy, "*Impact of Continuous Integration on Code Reviews*", In Proceeding of The 14<sup>th</sup> International Conference on Mining Software Repositories (**MSR 2017**), pp. 499–502, Buenos Aires, Argentina, May, 2017
- (14) **M. Masudur Rahman**, C. K. Roy, and Jason Collins, "*CORRECT: Code Reviewer Recommendation in GitHub Based on Cross-Project and Technology Experience*", In Proceeding of The 38<sup>th</sup> International Conference on Software Engineering (Companion Volume) (**ICSE 2016**), pp. 222–231, Austin Texas, USA, May 2016 (Acceptance rate: **28/108=26.00%**)

- (15) **M. Masudur Rahman** and C. K. Roy, “*QUICKAR: Automatic Query Reformulation for Concept Location Using Crowdsourced Knowledge*”, In Proceeding of The 31<sup>st</sup> IEEE/ACM International Conference on Automated Software Engineering (ASE 2016) (New Ideas Track), pp. 220–225, Singapore, September 2016
- (16) **M. Masudur Rahman**, C. K. Roy, Jesse Redl, and Jason Collins, “*CORRECT: Code Reviewer Recommendation at GitHub for Vendasta Technologies*”, In Proceeding of The 31<sup>st</sup> IEEE/ACM International Conference on Automated Software Engineering (ASE 2016) (Tool Demo Track), pp. 792–797, Singapore, September 2016
- (17) **M. Masudur Rahman**, C. K. Roy and David Lo, “*RACK: Automatic API Recommendation using Crowdsourced Knowledge*”, In Proceeding of The 23<sup>rd</sup> IEEE International Conference on Software Analysis, Evolution, and Reengineering (SANER 2016), pp. 349–359, Osaka, Japan, March 2016 (Acceptance rate:  $52/140=37.00\%$ )
- (18) Amit K. Mondal, **M. Masudur Rahman** and C. K. Roy, “*Embedded Emotion-based Classification of Stack Overflow Questions Towards the Question Quality Prediction*”, In Proceeding of The 28<sup>th</sup> International Conference on Software Engineering & Knowledge Engineering (SEKE 2016), pp. 521–526, San Francisco Bay, California, USA, July 2016
- (19) **M. Masudur Rahman**, C. K. Roy and Iman Keivanloo, “*Recommending Insightful Comments for Source Code using Crowdsourced Knowledge*”, In Proceeding of The 15<sup>th</sup> IEEE International Working Conference on Source Code Analysis and Manipulation (SCAM 2015), pp. 81–90, Bremen, Germany, September 2015 (Acceptance:  $24/68=35.00\%$ )
- (20) **M. Masudur Rahman** and C. K. Roy, “*Recommending Relevant Sections from a Webpage about Programming Errors and Exceptions*”, In Proceeding of The 25<sup>th</sup> International Conference on Computer Science and Software Engineering (CASCON 2015), pp. 181–190, Markham, Canada, November 2015 (Acceptance rate:  $21/71=29.57\%$ )
- (21) **M. Masudur Rahman** and C. K. Roy, “*An Insight into the Unresolved Questions at Stack Overflow*”, In Proceeding of the 12<sup>th</sup> Working Conference on Mining Software Repositories (Challenge Track) (MSR 2015), pp. 426–429, Florence, Italy, May 2015
- (22) **M. Masudur Rahman** and C. K. Roy, “*TextRank Based Search Term Identification for Software Change Tasks*”, In Proceeding of the 22<sup>nd</sup> IEEE International Conference on Software Analysis, Evolution, and Reengineering (ERA Track) (SANER 2015), pp. 540–544, Montreal, Canada, March 2015
- (23) **M. Masudur Rahman**, S. Yeasmin and C. K. Roy, “*Towards a Context-Aware Meta Search Engine for IDE-Based Recommendation about Programming Errors and Exceptions*”, In Proceeding of the

IEEE CSMR-18/WCRE-21 (CSMR/WCRE 2014), pp. 194–203, Antwerp, Belgium, February 2014  
(Acceptance rate: **27/87=31.00%**)

- (24) **M. Masudur Rahman** and C. K. Roy, “*On the Use of Context in Recommending Exception Handling Code Examples*”, In Proceeding of the 14<sup>th</sup> IEEE International Working Conference on Source Code Analysis and Manipulation (SCAM 2014), pp. 285–294, Victoria, Canada, September 2014 (Acceptance rate: **26/82=31.70%**)
- (25) **M. Masudur Rahman** and C. K. Roy, “*SurfClipse: Context-Aware Meta Search in the IDE*”, In Proceeding of the 30<sup>th</sup> International Conference on Software Maintenance and Evolution (Demo Track) (ICSME 2014), pp. 617–620, Victoria, Canada, September 2014
- (26) **M. Masudur Rahman** and C. K. Roy, “*An Insight into the Pull Requests of GitHub*”, In Proceeding of the 11<sup>th</sup> Working Conference on Mining Software Repositories (Challenge Track) (MSR 2014), pp. 364–367, Hyderabad, India, May 2014
- (27) **M. Masudur Rahman**, S. Yeasmin and C. K. Roy, “*An IDE-Based Context-Aware Meta Search Engine*”, In Proceedings of the 20<sup>th</sup> Working Conference on Reverse Engineering (ERA Track) (WCRE 2013), pp. 467–471, Koblenz, Germany, October 2013

## 1.5 Outline of the Thesis

The thesis contains nine chapters in total. In order to deal with query reformulation challenges in code search, we conduct six independent but interrelated studies. Our studies target three maintenance task contexts – *concept location*, *bug localization* and *Internet-scale code search*. This section outlines different chapters of the thesis as follows.

- **Chapter 2** provides a background overview on automated search query reformulations such as types/steps of reformulation, working contexts and implications of reformulation.
- **Chapter 3** discusses the first study namely **STRICT** that accepts a change request as a query and delivers a reformulated query for concept location.
- **Chapter 4** presents the second study namely **ACER** that reformulates a given query for concept location with appropriate keywords extracted from the relevant source code.
- **Chapter 5** presents the third study namely **BLIZZARD** that accepts either a noisy or poor bug report as a query, and delivers a reformulated query for bug localization.
- **Chapter 6** discusses the fourth study namely **BLADER** that reformulates a given query for bug localization using clustering tendency analysis and word embeddings technology.

- **Chapter 7** presents **RACK** that accepts a generic query on a programming task, and reformulates the query with relevant API classes mined from Stack Overflow for Internet-scale code search.
- **Chapter 8** presents **NL2API** that reformulates a given query for Internet-scale code search using crowd generated knowledge from Stack Overflow and word embeddings technology.
- **Chapter 9** concludes the thesis with a list of future research directions inspired by this PhD thesis.

# CHAPTER 2

## BACKGROUND

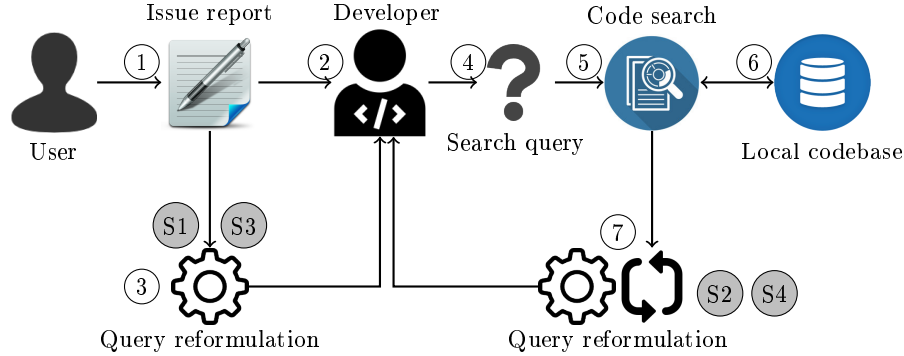
In this chapter, we introduce the terminologies and background concepts that are required to follow the rest of thesis. Section 2.1 defines automated query reformulation, Section 2.2 discusses the types of reformulation and Section 2.3 describes the working contexts of query reformulation. Section 2.4 focuses on the steps, Section 2.5 discusses term weighting algorithms, and Section 2.6 explains the implications of automated query reformulations. Section 2.7 explains word embeddings, Section 2.8 defines cosine similarity, and finally, Section 2.9 summarizes this chapter.

### 2.1 Automated Query Reformulation

Software developers search for source code both in a local codebase (e.g., Fig. 2.1) and in the Internet-scale code repositories (e.g., Fig. 2.2). One primary step of any kind of search operation is the construction of an appropriate query that reflects the information need. Unfortunately, developers often fail to choose the right queries for code search. Existing studies [120, 125] show that they might fail even 88% of the time. As a result, they need to frequently modify their queries (1) by adding new keywords, (2) by removing poor keywords or (3) by replacing the existing keywords with more appropriate ones. When these reformulations are performed using tool supports, they are called *automated query reformulations*. Although introduced by the Information Retrieval (IR) community several decades ago, automated query reformulation has been an active research area within Software Engineering for more than a decade [82, 146, 226].

### 2.2 Types of Query Reformulation

Constructing an appropriate search query that reflects one’s information need has always been a challenging task [60]. The task is even more challenging for source code search. Search queries often comprise of natural language (NL) keywords. On the contrary, source code documents are a mix of programming language keywords, identifier names and complex programming constructs [121]. That means, search queries and source code documents deal with two different vocabularies which have a little overlap and possibly different semantics [94, 233]. Thus, search queries chosen by the developers often could be either poor or inappropriate for the source code search. Different types of queries thus require different types of reformulations [98]. Traditionally, query reformulations are classified into three major categories as follows.



**Figure 2.1:** Automatic query reformulations in local code search

### 2.2.1 Query Expansion

A given search query is expanded by adding similar (e.g., synonyms) or complementary keywords. This is the most common reformulation strategy since the initial queries from the developers are often short. Studies [78, 219] show that an average search query contains 1–3 keywords. According to existing findings, 33%–76% of the queries are incrementally expanded by the developers during code search on the web [45, 219]. Majority of our proposed solutions in this thesis [182, 189, 194, 206] also perform query expansion.

### 2.2.2 Query Reduction

A given search query is refined by removing the noisy, ambiguous or less discriminative keywords from the query. One of the widely used heuristics for query reduction is *document coverage*. That is, keywords that are found in more than 25% of the documents within a corpus are removed from the query. These keywords are not specific enough and thus might fail to retrieve the documents of interest [98, 188]. Earlier studies also select the best sub-query as a form of query reduction [127]. A few other studies also retain important keywords from a given search query by using term weighting methods [120] and POS tagging [271, 273].

### 2.2.3 Query Replacement

The keywords of a given query are replaced with more appropriate ones [47, 129, 139, 252, 261]. Such a replacement could be intended for spelling corrections [85], query generalization or for query specialization [144, 157]. Developers often learn new information from browsing the search results, redefine their information needs, and then replace the initial search query altogether with more appropriate ones [87, 151].

## 2.3 Working Contexts of Query Reformulation

Working context often plays a key role in automated tool supports. Existing studies report significant benefits of incorporating contextual information in the automation of several Software Engineering tasks such as web search [70, 198], exception handling [184] and code snippet search [86, 107, 184]. Similarly, automated

reformulation of a search query could be guided by its contexts. Based on the developer’s working context, source code searches and their query reformulations can be classified in two broad categories as follows.

### 2.3.1 Local Code Search

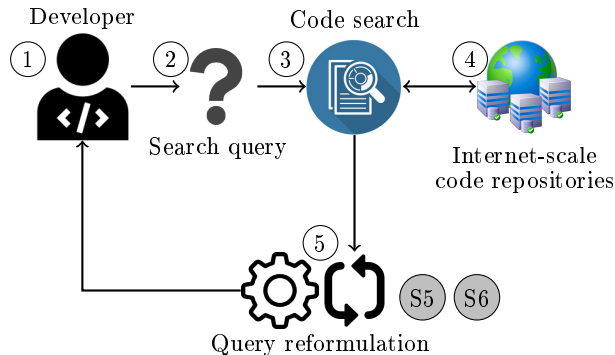
Local code search is a primary step of several software maintenance tasks such as *bug localization*, *concept location*, and *feature location*. Each of these maintenance tasks is initiated by a software user through his/her submission of an issue report (a.k.a., bug report, change request, feature request) (i.e., Step 1, Fig. 2.1). A developer then (1) makes use of the report texts, (2) constructs one or more queries, and (3) then searches for the source code locations that need to be repaired or enhanced. This search is limited within a local codebase (i.e., single software system). Query reformulation approaches supporting this code search mostly analyse the issue report texts and the local codebase (i.e., Steps 1, 3, 6, 7, Fig. 2.1) to deliver the reformulated queries. Local code searches can be divided into three major categories as follows:

- **Bug Localization** localizes a hidden *software bug* or a *defect* within the source code of a system [276]. Existing studies [130, 170, 250] adopt several methodologies – spectral analysis, static analysis and Information Retrieval – for bug localization. In this thesis, we deal with Information Retrieval (IR)-based bug localization since it has a high potential for low cost debugging [207, 248]. IR-based localization leverages the textual similarity between a bug report and the source code. Bug reports verbatim often do not make good search queries. Hence, they need to be reformulated carefully before using them in the IR-based bug localization [231].
- **Feature Location** finds out a *software feature* that is implemented within the source code of a software system [69]. Features are the visible components or attributes of a system that the users can interact with. Feature location deals with feature requests [178].
- **Concept Location** finds out an *abstract programming concept* (e.g., linked list) that is implemented within the source code of a software system [98]. All concepts implemented in the code might not result into visible software features. Both bug localization and feature location can be considered as two special cases of concept location.

### 2.3.2 Internet-Scale Code Search

Code search on the web is an integral part of problem solving activities of the software developers. It is initiated by a developer through his/her submission of a free-form natural language (NL) query to a code search engine (e.g., GitHub code search). Recent studies [205, 219] suggest that developers also frequently use general-purpose web search engines (e.g., Google) for code search. Unlike local code search (limited to single code repository), this search is performed over thousands of code repositories across various application domains (Steps 3, 4, 5, Fig. 2.2). That is, the code corpus is much bigger and noisier. Hence, construction of an appropriate query is a major challenge for this search. Besides, supporting materials such as issue reports might not be available in this case. Several studies attempt to discover the intent [110, 201, 274], linguistic





**Figure 2.2:** Automatic query reformulations in the Internet-scale code search

or semantic issues [144, 234] of a free-form NL query, and then reformulate the query. The goal is to retrieve one or more relevant code examples that meet the information need expressed in the query.

## 2.4 Steps of Automated Query Reformulation

The steps of automated query reformulation might vary based on the type or working context of a search query. However, most of the existing approaches from the literature [84, 96, 98, 189, 191, 231] share a common set of steps. These steps can be categorized into three major tasks as follows.

### 2.4.1 Query Feedback Collection

The first step towards improving a given query is to collect feedback on the query from reliable sources (e.g., developers). A number of studies from the Information Retrieval domain investigate the notion of *relevance feedback* in query reformulation. Rocchio [213] first introduced relevance feedback in the context of Vector Space Model (VSM) back in 1971, which was adopted by dozens of later studies. Lucia et al. [146] first apply relevance feedback in the context of Software Engineering where they deal with traceability link recovery problem. Then later studies incorporate that into concept/concern location [84, 98, 105] and bug localization [231]. The underlying idea is to collect meaningful, reliable feedback on a given query and then to leverage such feedback opportunistically in improving the query. Relevance feedback mechanism supporting query reformulations can be classified into three major categories as follows.

**(a) Explicit Relevance Feedback:** Developers are expected to provide *explicit* feedback on the relevance of the documents retrieved by a given query. They annotate each of these documents as either *relevant* or *irrelevant* to their information need [84, 251]. Although these feedback could be accurate and meaningful, capturing them regularly from the developers is time-consuming and sometimes even impossible.

**(b) Implicit Relevance Feedback:** Given the high cost of explicit feedback, several studies focus on capturing low cost feedback that is implicitly provided by the developers [119]. This type of feedback comprises of developer’s reactions towards the retrieved documents such as eye movements, document examination patterns, and keyword deletion or retention patterns.

(c) **Pseudo-Relevance Feedback:** Unlike the above two types, this feedback does not warrant for developer intervention. It is often a part of fully automated query reformulation. That is, the Top-K source documents retrieved by a given query are naively considered as *relevant* to the query [98, 104, 222]. It is also known as *blind feedback*. Existing studies from the literature [98, 188, 189, 192, 222] have provided significant evidence that such feedback is indeed useful for reformulating and improving a search query.

## 2.4.2 Candidate Keyword Selection

Once the feedback on a query is collected, the next step is to identify appropriate candidate keywords for query reformulation with the help of such feedback. Existing approaches [98, 191, 231] attempt to find out appropriate candidate keywords from various information sources including the relevance feedback documents. In order to do that, they often make use of various term weighting methods borrowed from the Information Retrieval (IR) domain. Term weighting methods can be roughly categorized into three major categories –(1) frequency-based methods, (2) graph-based methods and (3) probabilistic methods. According to our systematic literature review, TF-IDF [114], a frequency-based method, has been extensively used for keyword selection and/or query reformulation in the context of code search.

## 2.4.3 Reformulation of a Search Query

Once candidate keywords are collected, they are ranked based on their estimated weights, and only Top-K (e.g., K=10) candidates are suggested [62, 191, 213, 231]. These important keywords are then used to expand or replace the given poor query. Haiduc et al. [98] argue that a single reformulation strategy might not be appropriate for all queries. That is, different queries might need different types of reformulations (e.g., expansion, reduction, replacement). Towards this goal, several studies [96, 98, 164, 189] adopt query difficulty analysis and machine learning to deliver the best reformulated query. In particular, they (1) construct multiple reformulation candidates for a given query, (2) determine their quality using 21–28 query difficulty metrics from the Information Retrieval domain [62], and then (3) deliver the best candidate as a reformulated query using machine learning.

## 2.5 Term Weighting

Determining the relative importance of a term/word within a body of texts is commonly known as *term weighting* [40, 101]. Although the underlying concepts and algorithms were introduced by the Information Retrieval (IR) community, term weighting has been frequently used by the IR-based solutions targeting Software Engineering problems (e.g., code search, bug localization). Two term weighting algorithms are frequently mentioned throughout the rest of this thesis as follows.

### 2.5.1 TF-IDF

Jones [114] proposed TF-IDF to determine the relative importance of a term within a body of texts (e.g., news article) back in 1972. TF-IDF stands for Term Frequency (TF)  $\times$  Inverse Document Frequency (IDF). While TF counts the occurrences of a term within a document, IDF is based on the multiplicative inverse of the number of documents in the corpus that contain the target term. TF-IDF can be calculated using different variants of TF and IDF as follows:

$$\text{TF-IDF}(t,d) = f_{t,d} \times \log\left(\frac{|D|}{n_t} + 0.01\right) \quad (2.1)$$

$$\text{TF-IDF}(t,d) = (1 + \log f_{t,d}) \times \log\left(1 + \frac{|D|}{n_t}\right) \quad (2.2)$$

Here  $f_{t,d}$  refers to the frequency of a term  $t$  in the document  $d$ ,  $n_t$  refers to the number of documents containing the term  $t$ , and  $D$  is the set of all documents in the corpus. Thus, if a term is frequent within a document but not so frequent in other documents across the corpus, then this term is considered to be *important* (e.g., search keyword) within the target document.

TF-IDF adopts the notion of *term independence*. That is, it does not capture the impact of the surrounding terms upon a given term while determining the importance of the given term. [137]. However, idioms and phrases clearly depend on each other for their comprehensive meaning. For example, “*search engine*”, a noun phrase, conveys a different semantic than that of “*search*” and “*engine*” in isolation. Besides, important keywords of a document might always not be the most frequent ones, which is especially observed with the source code documents [102]. Despite these issues above, many of the existing studies [62, 98, 213] adopt frequency based term weights (e.g., TF-IDF) in keyword selection (and query reformulation) since they are light weight, intuitive and easy to use.

### 2.5.2 TextRank & POSRank

Unlike TF-IDF [114], several existing algorithms capture dependencies among the terms within a body of texts using term co-occurrences [153] and syntactic dependencies [53]. First, they transform each text document into a graph where the nodes represent the distinct words and the connecting edges refer to the dependencies among the words from the document. Second, they adapt Google’s PageRank algorithm [57] for natural language texts, and determine the relative weight of each word using recursive weight computation (e.g., Chapter 3). Term weight based on word co-occurrences is called TextRank whereas the weight computed using syntactic dependencies is called POSRank [53]. Several of our studies [189, 191, 192] leverage these dependencies among the words for determining their relative importance. Unlike regular texts (e.g., news articles), source code is often scarce in vocabulary but rich in structures and dependencies [102]. Thus, graph-based algorithms could be a more suitable choice than TF-IDF for term weighting and for candidate

term selection from the source code. Our experiments [189, 192] also support this propositions by providing positive empirical evidence.

## 2.6 Implications of Automated Query Reformulation

Developers often fail to choose the appropriate queries from a change request while locating the concepts within a software system [120]. As a result, they need to reformulate their queries frequently [98]. The same challenge also exists in the Internet-scale code search, which warrants frequent query reformulations [205, 219]. In short, constructing an appropriate query is challenging regardless of the working context of a code search operation. Thus, automated supports are highly warranted in query construction for the low cost code searches. However, like any other automated tool supports, automated query reformulation comes with both positive and negative implications as follows.

### 2.6.1 Benefits of Query Reformulation

Traditional web/code search engines encourage short and concise queries [205, 263]. Developers thus often use short queries (e.g., 1-3 keywords) for code search which might not reflect their information need properly [78, 219]. There is also a little chance (i.e., 10%–15%) that a developer might guess the exact same words used in the source code documents that were authored by other developers [83]. All these circumstances make the appropriate query construction highly challenging. Fortunately, addition of similar (e.g., synonyms) or complementary keywords often improves a poorly designed query. Existing studies have reported  $\approx 20\%$  performance improvement as a result of automated query reformulations [145]. Furthermore, automatically reformulated queries and the documents retrieved by them help the developers (1) redefine their information needs and (2) retrieve the source code documents of interest more quickly.

### 2.6.2 Costs of Query Reformulation

While query reformulations are useful, they might have a few negative effects as well. For example, automated reformulations might hurt such queries that are already of high quality [61, 98, 189]. Adding extra keywords to these queries makes them noisy. Hence, they might drift away from their original topics. Existing studies [172, 228, 233] also argue that inappropriate reformulations of a search query are more harmful than no reformulation at all. There also exist a few queries in each subject system namely *difficult queries* which could not be improved using the existing approaches from literature [61, 87].

Given these two-fold implications of automated query reformulation above, contemporary approaches attempt to maximize the benefits and minimize the costs of reformulation. For example, several studies [96, 98, 189] adopt machine learning techniques and query difficulty analysis to improve the poor queries and to preserve the high quality queries during query reformulation.

## 2.7 Word Embeddings

Traditional code search engines (e.g., Lucene, GitHub code search) often suffer from vocabulary mismatch issues (e.g., polysemy, synonymy) [95, 142, 265]. One crucial step towards tackling these challenges is to determine the semantics of a word correctly. There have been several studies [156, 188, 268, 272] that attempt to determine the semantics of a word using its context which is captured from a large corpus (e.g., Stack Overflow). Mikolov et al. [156] propose *word2vec*, a feed-forward neural network based text mining tool that mines a corpus and represents each word as a high dimensional numeric vector. This vector is also called *word embeddings* [156, 268]. In order to learn embeddings, *word2vec* uses two predictive models— continuous bag of words (CBOW) and skip-gram. CBOW model predicts a word given its contextual words whereas skip-gram attempts to predict the context of a given word. Embeddings are learned in such a way that similar words co-occur close to each other within a high dimensional semantic space. Two of our conducted studies (Chapters 6, 8) make use of word embeddings in reformulating queries for source code search. In this thesis, we use an updated version of *word2vec* namely *fastText* [54] and Stack Overflow as a corpus for learning our word embeddings [194].

## 2.8 Cosine Similarity

It is a measure<sup>1</sup> that indicates the orientation between two vector spaces with varying number of dimensions. Cosine similarity is frequently used in Information Retrieval to determine the lexical similarity between any two text documents. In particular, each unique term is considered as a dimension and each document is considered as a vector of such dimensions. Let us consider that  $A$  and  $B$  are two documents representing a query and a code segment respectively in our research context. First,  $A$  and  $B$  are normalized using standard natural language preprocessing (e.g., stop word removal, punctuation removal, token splitting), and a combined vector  $C$  is constructed using all the unique terms from them. Then the cosine similarity  $S_{cos}$  between  $A$  and  $B$  can be calculated as follows.

$$S_{cos} = \frac{\sum_{i=1}^n A_i \times B_i}{\sqrt{\sum_{i=1}^n A_i^2} \times \sqrt{\sum_{i=1}^n B_i^2}} \quad (2.3)$$

Here,  $A_i$  represents the weight (e.g., TF, TF-IDF) of  $i^{th}$  term from  $C$  in vector  $A$ , and  $B_i$  represents the similar weight in vector  $B$ . This measure values from zero (i.e., complete dissimilarity) to one (i.e., complete lexical similarity). The measure is widely used, intuitive and easy to calculate.

In this thesis, we use this measure to determine the *semantic distance* between any two words that are represented as embedding vectors. We also use Lucene [32], a cosine similarity based code search engine, to retrieve the relevant source code against original and reformulated search queries.

---

<sup>1</sup><https://bit.ly/KTDKUy>

## 2.9 Summary

In this chapter, we introduced several important terminologies and background concepts that would help one to follow the remaining of thesis. We defined automated query reformulation, and discussed its three types and two working contexts– *local code search* (e.g., concept location, bug localization) and *Internet-scale code search*. We discussed the common steps of query reformulation, term weighting algorithms (e.g., TF-IDF, TextRank) and explained both positive and negative impacts of automated query reformulations. Finally, we defined word embeddings and cosine similarity which are frequently used throughout this thesis. In the next six chapters, we discuss our six studies supporting concept location (Chapters 3, 4), bug localization (Chapters 5, 6) and Internet-scale code search (Chapters 7, 8) with automated query reformulations. While Fig. 2.1 shows the working contexts of the first four studies (S1, S2, S3, S4), Fig. 2.2 does the same for the remaining two studies (S5, S6).

## CHAPTER 3

# SEARCH QUERY REFORMULATION FOR CONCEPT LOCATION USING GRAPH-BASED TERM WEIGHTING

Software maintenance costs about 60% of the total development time and efforts [88]. Developers receive thousands of software change requests from the users during maintenance phase. Change requests are often a mix of unstructured regular texts and domain level concepts. Developers need to find the right keywords from these concepts so that they could identify the relevant code entities (that should be changed) using a search technique. Unfortunately, according to existing evidence [120], choosing the right search keywords is highly challenging even for the experienced developers. In this chapter, we attempt to overcome this keyword selection challenge. Here, we present our first study (STRICT) that accepts a change request as a search query, identifies the appropriate search keywords from the request texts, and then delivers an improved, reformulated search query for the concept location task.

The rest of the chapter is organized as follows: Section 3.1 presents a brief overview of our study and Section 3.2 offers a motivating example. Section 3.3 presents our proposed technique for search query construction intended for the concept location. Section 3.4 discusses our experiments, results and validation, Section 3.5 identifies the threats to validity, Section 3.6 discusses the related work, and finally Section 3.7 concludes the chapter with future work.

### 3.1 Introduction

During software maintenance, developers deal with a number of change requests. They make frequent changes to the source code of a software system in order to address these requests. Before making a code change, one needs to identify the exact locations in the code, which is a major challenge, even for a medium sized system [265]. Such challenge is exacerbated by the unstructured nature of a submitted request. Change requests are often written by the users of a software system who might be familiar with the application domain (e.g., word processing) of a software (e.g., Microsoft Word). However, they generally lack the idea of how a particular software feature is implemented in the software code. Hence, a change request from them generally entails unstructured natural language texts and one or more *“high level”* concepts from the application domain. A developer needs to map these *concepts* to the relevant code *locations* of a software system in order to

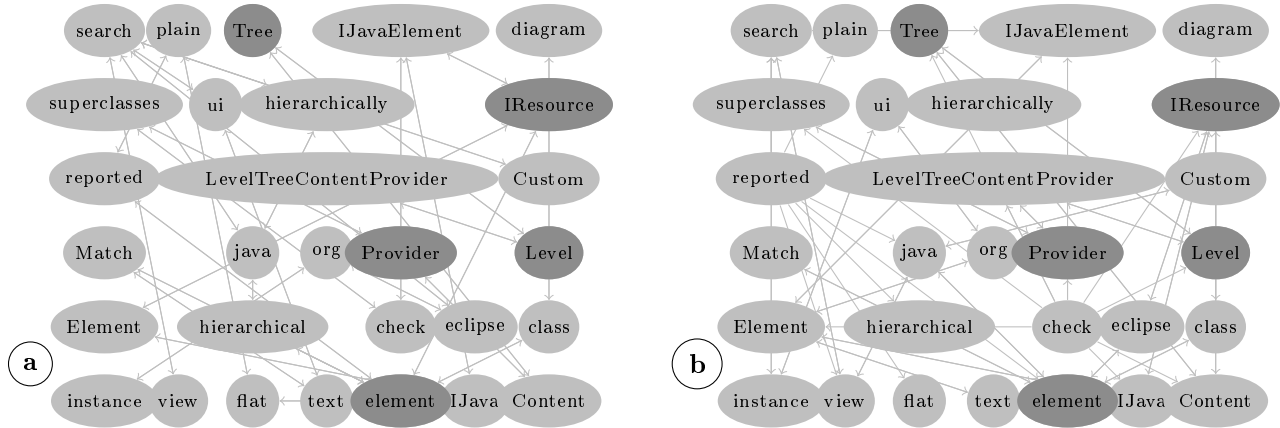
implement the changes requested by the users [121, 149]. Such mapping has been termed as *concept location* or *concept assignment problem* by the research community over the years [84, 98, 142].

The mapping between concepts and source code is possibly trivial for a developer who has substantial knowledge on a target system. Unfortunately, developers involved in the maintenance might always not be aware of the low-level architecture of a software system. The design documents required for a code change might also not be available [72, 171]. Thus, the developers often experience difficulties in identifying the exact source code locations (e.g., methods) that need to be changed. The conceptual mapping above generally starts with a search within the project codebase which requires one or more suitable search terms [98]. Unfortunately, the developers perform poorly in choosing appropriate queries from a change request regardless of their development experience [83, 120, 142]. Based on a user study, Kevic and Fritz [120] report that only 12.20% of the search terms chosen by the developers were able to retrieve the code of interest. Furthermore, the chances that a developer would correctly guess the exact words used in the source code are slim (i.e., 10% – 15%) [83]. Therefore, search term identification for concept location is a major challenge for the developers. One way to help them overcome this challenge is to automatically suggest suitable search terms from the change request texts at hand. Our work in this chapter addresses this particular research problem—*search term suggestion*—for concept location.

Relevant existing approaches from the literature apply lightweight heuristics [120], relevance feedback [84, 93, 95, 98, 188], query difficulty analysis [93, 95, 97, 98], natural language processing [65, 104, 226] and software repository mining [104, 109, 265]. However, most of these approaches expect a developer to provide the initial search query which they can improve upon. Unfortunately, preparing such a query is often a non-trivial task for the developers as shown by the existing evidence [83, 120, 142]. One can think of using the whole texts (i.e., *title* + *description*) of a change request as a search query. However, such texts often produce verbose and poor queries [64, 120]. Kevic and Fritz propose the only approach for automatically identifying search terms from a change request. They consider several heuristics concerning *frequency*, *location*, *part of speech* and *notation* of the terms from a request text. According to preliminary evaluation, their model is found promising. However, it suffers from two major limitations. First, their model is neither trained using a large dataset nor cross-validated against multiple software systems. They use a small dataset of only 20 change requests from a single subject system. Since the dataset was small and restricted to one system only, their model might require frequent re-training for other set of bug reports. Thus, their model is yet to be matured and reliable. Second, TF-IDF is reported as the most important feature of their model. However, TF-IDF fails to capture the semantic or syntactic dependencies among terms during the estimation of term importance, which has been reported as its major limitation [53, 153].

In this chapter, we propose and design a novel query suggestion technique—*STRICT*—that automatically identifies and recommends high quality search terms from a change request for concept location. We first determine the importance of each term from the texts of a request by employing three graph-based term weighting algorithms—*TextRank*, *POSRank* and *WK-Core*, and then suggest the most important terms as a





**Figure 3.1:** Text graphs of the change request in Table 3.1 – (a) using word co-occurrences, and (b) using syntactic dependencies

search query. Both TextRank and POSRank are the adaptations of *PageRank* algorithm [57] in the context of natural language texts. They first transform the textual content of a change request into a text graph (e.g., Fig. 3.1) by using either co-occurrences or syntactic dependencies among the terms. Then they determine term importance by leveraging the topology of the constructed graphs with PageRank algorithm [101, 153]. WK-Core extracts a cohesive sub-graph from such a text graph using K-Core decomposition, and then identifies the important terms using centrality and cohesion of the nodes in the sub-graph. Unlike the model of Kevic and Fritz [120] that overlooks term dependencies, our term weighting methods leverage term dependencies (e.g., semantic & syntactic dependencies) and textual cohesion within a change request for determining the term importance [53, 153, 217]. Thus, our approach has a higher potential for returning the good quality search terms from a given change request.

Experiments using 2,885 change requests from eight Java-based subject systems report that our technique—STRICT—can provide higher quality search terms than 43%–74% of the baseline queries ( $\mathbf{RQ}_1$ ) which is highly promising according to the literature [98, 164]. Our suggested queries can retrieve relevant source code documents for 46%–78% of the change tasks with 29% precision and a reciprocal rank of 0.29 which are 26%, 25% and 26% higher respectively than the best performing baseline ( $\mathbf{RQ}_2$ ). Comparisons with two state-of-the-art techniques—Kevic and Fritz [120] and Rocchio [213]—report that our technique can improve 19%–23% more of the baseline queries than the state-of-the-art ( $\mathbf{RQ}_5$ ). Furthermore, our queries achieve 37% higher accuracy, 31% higher precision and 38% higher reciprocal rank than the state-of-the-art when Top-10 results are analysed ( $\mathbf{RQ}_6$ ). All these findings above clearly demonstrate the high potential of our approach over the state-of-the-art.

**Novelty of Contributions:** Our work in this chapter is a significantly extended version of our earlier work on search term identification [191]. The earlier work (1) proposes a basic graph-based term selection approach, and (2) conducts evaluation using a limited set of 1,939 change requests and four research questions.

**Table 3.1:** An Example Change Request (Issue #303705, eclipse.jdt.ui)

Field	Content	QE
Title	[search] Custom search results not shown hierarchically in the java search results view	559
Description	Consider an instance of <code>org.eclipse.search.ui.text.Match</code> with an element that is neither an <code>IResource</code> nor an <code>IJavaElement</code> . It might be an element in a class diagram, for example. When such an element is reported, it will be shown as a plain, flat element in the otherwise hierarchical java search results view. This is because the <code>LevelTreeContentProvider</code> and its superclasses only check for <code>IJavaElement</code> and <code>IResource</code> .	63
<b>An Example of Query Suggestion</b>		
Baseline	{Title + Description}	14
<b>STRICT</b>	{ <i>element IResource Provider Level Tree</i> }	<b>01</b>

**QE** = Rank of the first correct result returned by the query

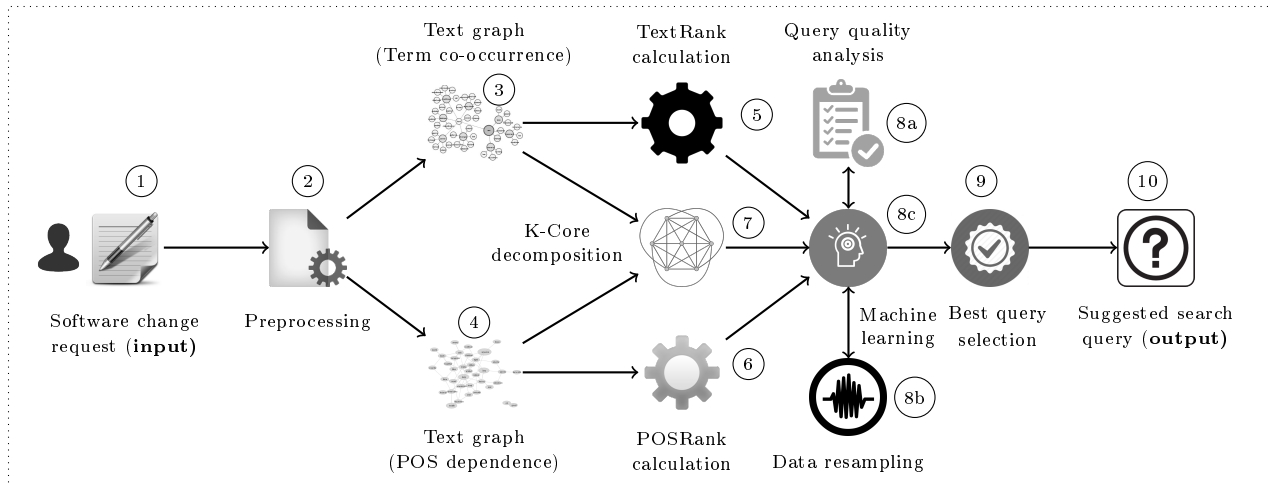
On the other hand, our work in this chapter provides an improved version of the approach using not only three graph-based algorithms (*TextRank* [153], *POSRank* [53], *Weighted K-Core* [217]) but also two novel aspects such as *query difficulty analysis* and *machine learning*. We also conduct a more extended evaluation using 2,885 change requests, and answer *seven* research questions. We perform more in-depth analysis for the research questions that are taken from our earlier work [191], and contrast between difficult and easy queries. Furthermore, we provide a *working prototype* and detailed replication package [24] which have been successfully verified by the third parties.

Thus, we make the following contributions in this work:

- (a) A novel query reformulation technique –**STRICT**– that accepts a change request as an input, and delivers a reformulated query (high quality search terms) for concept location.
- (b) Comprehensive evaluation of the technique using  $\approx 3\mathbf{K}$  change requests from eight Java-based subject systems, four state-of-the-art performance metrics and two different performance dimensions.
- (c) Comprehensive validation of the technique using comparisons with two state-of-the-art techniques on query construction [120, 213].
- (d) A verified replication package [24] that includes a working prototype, experimental data and other associated materials for replication and third party reuses.

## 3.2 Motivating Example

In order to demonstrate the capability of our approach in the search query suggestion, we provide an example where our query outperforms three baseline queries. Table 3.1 shows a change request from `eclipse.jdt.ui`



**Figure 3.2:** Schematic diagram of the proposed query reformulation technique—STRICT

system that reports a concern about custom search result display in Eclipse IDE. Our technique—STRICT—first transforms the textual content of the request into two text graphs by capturing (a) co-occurrences among the terms (i.e., Fig. 3.1-(a)) and (b) syntactic dependencies among the terms (i.e., Fig. 3.1-(b)) respectively. Then, it identifies the most important terms by recursively analysing the topological characteristics of both graphs and by employing term weighting, query quality analysis and machine learning. STRICT returns the following Top-5 search terms (i.e., highlighted, Fig. 3.1)—‘*element*’, ‘*IResource*’, ‘*Provider*’, ‘*Level*’ and ‘*Tree*’—which return the first correct result at the topmost position of the result list. On the contrary, the baseline queries—*Title*, *Description* and *Title+Description*—return the same result at the 559<sup>th</sup>, 63<sup>rd</sup> and 14<sup>th</sup> positions. Thus, our search query (1) can locate a starting point within the source code (for the code change) more easily, and thus, (2) can potentially reduce the manual efforts spent on query preparation, code search and on overall code changes by the developers.

### 3.3 STRICT: Automated Search Query Suggestion from a Change Request for Concept Location

Since appropriate search term identification is a major challenge for the developers, we introduce a novel approach for search term identification and suggestion from a software change request. Figure 3.2 shows the schematic diagram of our proposed technique—STRICT. Furthermore, Algorithms 1–3 present the pseudo-code of our approach. We first transform a change request into two text graphs (e.g., Fig. 3.1) based on word co-occurrences and syntactic dependencies among the words, and then identify suitable search terms using three term weighting algorithms, query difficulty analysis and machine learning as follows:

### 3.3.1 Data Collection

Our technique accepts the user-provided texts from a change request as the input (i.e., Step 1, Fig. 3.2), and returns a ranked list of search terms as the output (i.e., Step 10, Fig. 3.2). We collect change requests from two popular bug tracking systems—*BugZilla* and *JIRA*. Each change request is submitted as a semi-structured report written using natural language texts, and it contains several fields such as *Issue ID* (e.g., 303705), *Product* (e.g., JDT), *Component* (e.g., UI), *Title* and *Description*. We extract the last two fields from each report for analysis, as was also done by the literature [98, 120]. *Title* summarizes a requested change task whereas *Description* contains detailed explanation of the task provided by the submitter using natural language texts.

### 3.3.2 Text Preprocessing

We analyse *Title* and *Description* fields of a software change request, and perform several preprocessing operations on them (i.e., Step 2, Fig. 3.2, Lines 3–6, Algorithm 1). We consider *sentence* as a logical unit of the request texts, and collect each of the sentences from both fields. Then we perform standard natural language preprocessing (i.e., removal of stop words, keywords and punctuation marks, and splitting of structured terms) on each of these sentences, and extract the candidate search terms. In particular, we remove stop words or keywords, and turn each structured artifact (e.g., `org.eclipse.ui.part`) into multiple technical terms (e.g., `org`, `eclipse`, `ui` and `part`) using token splitting. We also split each-camel case token (e.g., `createPartControl`) into simpler tokens (i.e., `create`, `Part` and `Control`), and keep both simpler and camel-case tokens for our analysis [75, 98]. It should be noted that we avoid term stemming (i.e., extracting root form of a given term) since it degrades the performance of our technique, as was also reported by several earlier studies [106, 120].

### 3.3.3 Text Graph Development

**Using Term Co-occurrence:** After the preprocessing step above, we get a list of sentences from each change request. Each preprocessed sentence comprises of an ordered list of candidate search terms. We use these sentences to transform the change request into a *text graph* (i.e., Step 3, Fig. 3.2, Line 8, Algorithm 1). In the text graph, unique terms are represented as nodes and the *co-occurrences* of terms within each sentence are denoted as connecting edges (e.g., Fig. 3.1-(a)). The underlying idea is that all the terms that co-occur in the texts within a fixed window have some level of semantic relatedness or dependencies among them [53, 153]. For example, if we consider the sentence—*“Custom search results not shown hierarchically in the java search results view”*—from the example request texts (i.e., Table 3.1), the preprocessed version forms an ordered list of terms—*“custom search hierarchically java search view.”* It should be noted that the transformed sentence contains several phrases such as *“custom search”* and *“search view”*. The terms in each of these phrases complement each other semantically, and convey an enriched semantic than their original

generic semantics. Thus, these terms are semantically dependent on each other for a comprehensive new meaning. Term co-occurrence information captures such dependencies in a statistical sense. We thus employ a *sliding window* of *window size = 2* (as recommended by Mihalcea and Tarau [153]), and derive the following term co-occurrence relationships:

$custom \longleftrightarrow search$ ,  $search \longleftrightarrow hierarchically$ ,  $hierarchically \longleftrightarrow java$ ,  $java \longleftrightarrow search$  and  $search \longleftrightarrow view$ .

Then these term relationships are represented as connecting edges among the corresponding nodes in the text graph (i.e., Fig. 3.1-(a)). Given that both terms depend on each other for their semantics, we represent each of the connections using bi-directional edges (e.g., Fig. 3.1-(a)).

**Using Syntactic Dependence:** Although term co-occurrence captures semantic dependencies among the terms through a statistical sense, it might always not be effective for term weight estimation. Another orthogonal approach to capture term importance could be syntactic dependencies among the terms. Jespersen [113] suggests that words from a sentence can be divided into three major ranks—*primary* (i.e., nouns), *secondary* (i.e., verbs, adjectives), and *tertiary* (i.e., adverbs), which is often called as *Jespersen’s Rank Theory of Three*. According to this theory, a word from a higher rank defines (i.e., modifies) another word from the same or lower ranks within a sentence. Thus, a noun can modify only another noun whereas a verb can modify another noun, verb or adjective but not an adverb. We leverage this principle for our term weight estimation, capture the grammatical modifications or dependencies among the words, and then represent such dependencies as directed edges in the text graph (i.e., Step 4, Fig. 3.2, Line 9, Algorithm 1). We first annotate each of the sentences from a change request using Stanford POS tagger [244], and then group them according to their Jespersen ranks. For instance, the example statement—“*element reported plain flat element hierarchical java search view*”—can be organized into two ranks—*primary* (“*search*”, “*view*”, “*java*”, “*element*”), and *secondary* (“*plain*”, “*flat*”, “*hierarchical*”, and “*reported*”). We derive the following relationships based on their syntactic dependencies.

$search \longleftrightarrow view$ ,  $view \longleftrightarrow java$ ,  $java \longleftrightarrow element$ ,  $reported \longrightarrow search$ ,  $reported \longrightarrow view$ ,  $reported \longrightarrow java$ ,  $reported \longrightarrow element$ ,  $reported \longrightarrow plain$ ,  $reported \longrightarrow flat$ ,  $reported \longrightarrow hierarchical$ ,

Then we encode the above relationships into connecting edges in the text graph (e.g., Fig. 3.1-(b)). It should be noted that these dependencies could be mutual or uni-directional.

### 3.3.4 TextRank (TR) Calculation

Once a text graph is developed based on term co-occurrences within the request texts, we treat the graph as a regular connected network. We apply a popular graph-based ranking algorithm namely TextRank [53, 153] to estimate the importance of the nodes (i.e., terms) in the graph (Step 5, Fig. 3.2, Lines 10–11, Algorithm 1). TextRank is an adaptation of *PageRank algorithm* which was proposed by Brin and Page [57] originally for web link analysis. TextRank analyses the connected neighbours and their weights for each term  $v_i$  in the graph recursively, and then calculates the term weight,  $TR(v_i)$ , as follows:

$$TR(v_i) = (1 - \phi) + \phi \sum_{j \in V(v_i)} \frac{TR(v_j)}{|V(v_j)|} \quad (0 \leq \phi \leq 1) \quad (3.1)$$

Here,  $V(v_j)$  and  $\phi$  denote node list connected to  $v_i$  and *damping factor* respectively. In the text graph (e.g., Fig. 3.1-(a)), co-occurrences among the terms are represented as bi-directional edges between the corresponding nodes. In the context of web surfing, damping factor,  $\phi$ , is considered as the probability of randomly choosing a web page by the surfer, and  $1 - \phi$  as the probability of jumping off that page. Mihalcea and Tarau [153] use a heuristic value of  $\phi = 0.85$  for natural language texts in the context of keyword extraction, and we also use the same value for our TextRank calculation. We initialize each of the terms in the graph with a default value of 0.25, and run an iterative version of the algorithm [57]. It should be noted that the initial value of a term does not affect its final score [153]. The computation iterates until the scores of all the terms converge below a certain threshold or it reaches the maximum iteration limit (i.e., 100 as suggested by Blanco and Lioma [53]). As Mihalcea and Tarau [153] suggest, we use a heuristic threshold of 0.0001 for the convergence checking of the scores.

TextRank adopts the underlying mechanism of a recommendation system where a term (e.g., “*Custom*”) recommends (i.e., votes) another term (e.g., “*search*”) if the second term complements the semantics of the first term in any way (e.g., “*custom search*”) [153]. The algorithm captures votes cast for a term by analysing its connected edges within the text graph (e.g., Fig. 3.1-(a)). It should be noted that the votes could be cast by other terms both from a local context (i.e., same sentence) and from the global context (i.e., entire request texts). Thus, the algorithm determines importance of a term using both its local and global contexts. Once the computation is over, each of the nodes of the graph is found with a final score. Such score is considered as the relative weight or relative importance of the corresponding term within the whole request texts.

### 3.3.5 POSRank (POSR) Calculation

While TextRank operates on a text graph based on word co-occurrences (e.g., Fig. 3.1-(a)), POSRank determines term-weight by analysing syntactic dependencies among the terms (e.g., Fig. 3.1-(b), Step 6, Fig. 3.2). POSRank is another adaptation of *PageRank* [57] for natural language texts. Similar to TextRank, it also analyses connectivity of each term in the graph but considers the links according to their directions. Incoming links and outgoing links of the term are treated differently. Incoming links represent votes cast for the term by other terms whereas the outgoing links represent the opposite. Thus, POSRank  $POSR(v_i)$  of each term  $v_i$  is calculated as follows:

$$POSR(v_i) = (1 - \phi) + \phi \sum_{j \in In(v_i)} \frac{POSR(v_j)}{|Out(v_j)|} \quad (0 \leq \phi \leq 1) \quad (3.2)$$

Here  $In(v_i)$  and  $Out(v_i)$  denote the node lists to which node  $v_i$  is connected to through incoming and outgoing links respectively. Since the underlying mechanism of PageRank-based algorithms is recommendation (i.e.,

votes) from other nodes of the graph, POSRank follows the suit of TextRank. That is, it determines the weight (i.e., importance) of a term by capturing and analysing the weights of the incoming links recursively. It should be noted that not only frequent votes but also the votes from other high scored nodes of the graph are essential for a node (i.e., term) to be highly scored (i.e., important). Given the similar topological properties (i.e., Fig. 3.1-(b)), we apply the same settings as of TextRank (Section 3.3.4). In particular, we apply the same damping factor ( $\phi$ ), iteration count, initial score, and convergence threshold for the POSRank calculation of each of the terms.

### 3.3.6 Weighted K-Core Calculation

Based on a user study, Rousseau and Vazirgiannis [217] report that keywords chosen by human subjects are generally phrases rather than single words. Although TextRank and POSRank employ topological properties of a graph constructed from texts, they might return a list of terms that are neither coherent nor comprehensive about the information need [217]. One way to possibly address this challenge is to analyse the K-core of the graph. K-core refers to a connected sub-graph of a graph where each node has a degree of at least size K. K-core decomposition has been applied in identifying highly connected groups within a large social network [50] and in extracting a coherent set of keywords from a body of texts [217]. Similarly, we employ a weighted version of K-core decomposition for identifying coherent search terms from a change request for concept location. In particular, we first extract K-core from each of the two graphs above by invoking Algorithm 2 (i.e., Step 7, Fig. 3.2, Lines 13–15, Algorithm 1). We iterate through all the nodes of a graph, and delete the nodes (and their edges) having a degree below  $K$  (i.e., Lines 1–6, Algorithm 2). This process continues until the graph is left with only such nodes that have a weighted degree greater than  $K$ . Then we calculate score,  $WK-Core(v_i)$ , of each node  $v_i$  based on its degree and weights of the connecting edges (i.e., Lines 16–18, Algorithm 1) as follows:

$$WK - Core(v_i) = \sum_{j \in V(v_i)} w(v_i, v_j) \quad (3.3)$$

Here,  $w(v_i, v_j)$  denotes the weight of a connecting edge between nodes  $v_i$  and  $v_j$ , and  $V(v_i)$  refers to all nodes directly connected to  $v_i$ . We perform K-core decomposition on both text graphs constructed above (e.g., Fig. 3.1-a, b). Thus, the weight of an edge is determined based on either co-occurrence frequency or grammatical modification frequency between the two terms connected.

### 3.3.7 Term Ranking and Candidate Query Selection

Once the scores are calculated, we rank the candidate search terms based on their TextRank, POSRank and WK-Core (i.e., Lines 20–22, Algorithm 1). Then we collect top scored  $X\%$  (e.g.,  $X = 33$ ) candidate terms from each of the ranked lists, and construct the candidate search queries (i.e., Line 23, Algorithm 1). It should be noted that we collect a varying number of terms from each change request given that the requests

---

**Algorithm 1** Search Keyword Identification with Graph-based Term Weighting

---

```
1: procedure STRICT( $CR$ ) ▷  $CR$ : change request
2:    $L \leftarrow \{\}$  ▷ list of search terms
3:   ▷ collecting task details from the change request
4:    $T \leftarrow \text{collectTitle}(CR)$ 
5:    $D \leftarrow \text{collectDescription}(CR)$ 
6:    $TD \leftarrow \text{preprocess}(\text{combine}(T, D))$ 
7:   ▷ developing text graphs from the task details
8:    $G_{COC} \leftarrow \text{developTGUsingCo-occurrence}(TD)$ 
9:    $G_{POS} \leftarrow \text{developTGUsingPOS-dependence}(TD)$ 
10:  ▷ calculating TextRank and POSRank
11:   $TR \leftarrow \text{calculateTR}(G_{COC})$ 
12:   $POSR \leftarrow \text{calculatePOSR}(G_{POS})$ 
13:  ▷ collecting K-core from the graphs
14:   $G_{KCOC} \leftarrow \text{extractK-Core}(G_{COC}, K)$ 
15:   $G_{KPOS} \leftarrow \text{extractK-Core}(G_{POS}, K)$ 
16:  ▷ calculating K-core scores
17:   $KCCOC \leftarrow \text{calculateWK-Core}(G_{KCOC})$ 
18:   $KCPOS \leftarrow \text{calculateWK-Core}(G_{KPOS})$ 
19:  ▷ getting candidate queries and their difficulties
20:  Let  $CTS \leftarrow \{TR, POSR, KCCOC, KCPOS\}$ 
21:  for CandidateQueryKey  $ckey \in CTS.keys$  do
22:     $sortedTerms \leftarrow \text{sortByScore}(CTS[ckey])$ 
23:     $CQ[ckey] \leftarrow \text{getTopXPercent}(sortedTerms) + T$ 
24:     $QD[ckey] \leftarrow \text{getQueryDifficulty}(CQ[ckey])$ 
25:  end for
26:  ▷ getting the best search query for change request
27:   $QD' \leftarrow \text{resampleWithReplacement}(QD)$ 
28:   $QDM \leftarrow \text{developQueryDifficultyModel}(QD')$ 
29:   $BQ \leftarrow \text{getBestCandidateQuery}(QDM, \{CQ \cup TD\})$ 
30:   $L \leftarrow \text{getTopKSearchTerms}(BQ)$ 
31:  return  $L$ 
32: end procedure
```

---



---

**Algorithm 2** K-Core Decomposition of a Graph

---

```
1: procedure EXTRACTK-CORE( $G, K$ )
2:   Let  $F \leftarrow G$ 
3:    $\triangleright$  extracting the K-Core
4:   while nodeExists( $x$ ) and wDegree $_F(x) < K$  do
5:      $F \leftarrow \text{delete}(x, F)$ 
6:   end while
7:   return  $F$ 
8: end procedure
```

---

are of varying lengths. Earlier studies reported better performances for varying sized queries [50, 153, 217]. Our experimental results also suggest that variable size is better in terms of retrieval performance than a fixed size of the query. We also found that terms from *Title* field of a change request are more salient than those from the *Description* field. Hence, we append the terms from *Title* to each of the candidate queries as well. Existing studies have shown that no single reformulation strategy [98, 99], information source [188, 189] or retrieval algorithm [164] is sufficient enough for all queries under study. Similarly, we conjecture that no single term weighting method could be sufficient enough for search term identification from all the change requests. We thus develop six candidate search queries using the three term weighting methods discussed above. Table 3.2 shows the candidate queries for the showcase change request (i.e., Table 3.1) based on TextRank, POSRank and WK-Core score.

### 3.3.8 Best Query Suggestion with Machine Learning

Once multiple candidate queries are constructed from a software change request, the next challenge is to identify the best one among them for suggesting to the developer. Query difficulty prediction has been an active area of research to the Information Retrieval (IR) community [61]. Recently, such idea has also been adopted successfully in the Software Engineering problems [96, 98]. In the same vein, we also perform query difficulty analysis and then apply machine learning for identifying and suggesting the best candidate query (i.e., Steps 8–10, Fig. 3.2) as follows:

**Query Difficulty Analysis:** Prediction of query difficulty or query quality had been an active avenue of research for the Information Retrieval community over the last few decades. Haiduc et al. [96] first introduce query difficulty analysis in the context of Software Engineering where they employ 21 pre-retrieval metrics as query difficulty predictors. Pre-retrieval metrics do not require document retrieval to predict the quality of a given query. They are lightweight and often computed using the information gathered during corpus indexing (e.g., Term Frequency, Inverse Document Frequency). We use 20 of their metrics in our problem context (i.e., Step 8a, Fig. 3.2, Line 24, Algorithm 1), and capture four quality aspects of a given query – *specificity*, *coherency*, *term relatedness* among the query terms, and *textual similarity* between the query and

---

**Algorithm 3** Best Candidate Query Selection

---

```
1: procedure GETBESTCANDIDATEQUERY( $QDM, CQ$ )
2:    $P \leftarrow \{\}$  ▷ query difficulty class predictions
3:    $C \leftarrow \{\}$  ▷ instance occurrence counts
4:   for QueryDiffModel  $QDM_i \in QDM$  do
5:     for CandidateQueryKey  $ckey \in CQ.keys$  do
6:        $ptemp \leftarrow \text{getPredictionForHigh}(ckey, QDM_i)$ 
7:        $P[ckey] \leftarrow P[ckey] + ptemp$ 
8:        $C[ckey] \leftarrow C[ckey] + 1$ 
9:     end for
10:  end for
11:  for CandidateQueryKey  $ckey \in CQ.keys$  do
12:     $P[ckey] \leftarrow P[ckey] / C[ckey]$ 
13:  end for
14:   $highKey \leftarrow \text{sortKeyByPrediction}(P)$ 
15:  return  $CQ[highKey]$ 
16: end procedure
```

---

the document corpus. We collect query difficulty estimates of six candidate queries and one baseline query (e.g., Table 3.2) for each of the change requests.

**Dataset Labelling:** We adopt a supervised machine learning approach for identifying the best query among the candidates. Once query quality estimates are collected, we annotate each of the candidate queries based on their quality. In particular, we determine the *Effectiveness* of each query, and classify the six candidates and the baseline query into three classes— “*high*”, “*medium*” and “*low*”. The candidate query that returns the first correct result at the closest position to the top of a result list is annotated as *high* and the vice versa as *low*. The candidates that return the results between these two extreme positions are annotated as *medium* quality queries. Thus, each of the instances in our training dataset has 20 query quality (or query difficulty) predictors and one assigned class label.

**Data Resampling with Bootstrapping:** Our goal is to suggest only the *high* quality candidate query to a developer for any given change request. Thus, the training dataset constructed above is inherently skewed for the task. Only one out of each seven instances (i.e., six candidates + one baseline query) in the dataset could be a true positive. We thus perform bootstrapping [116, 237] on the training dataset with 100% sample size and with sample replacement (i.e., Step 8b, Fig. 3.2, Line 27, Algorithm 1). In particular, we make use of WEKA<sup>1</sup>, resample the training data multiple times (e.g., 100) and then develop multiple

---

<sup>1</sup><http://www.cs.waikato.ac.nz/ml/weka>

training datasets. The underlying idea was to introduce limited bias in the training dataset deliberately through resampling and thus to counteract the data skewness.

**Suggestion of the Best Candidate Query:** Once training datasets are ready, we apply machine learning on each of them, and develop a query difficulty model for each (i.e., Line 28, Algorithm 1, Step 8c, Fig. 3.2). Haiduc et al. [98] first employ Classification and Regression Tree (CART) algorithm for learning such model. However, RandomForest performs higher than CART according to our investigations. We thus use RandomForest for learning the query difficulty model in our work. Given the inherent data skewness and the limited strength of individual predictors, single quality model might be not sufficient enough for the best query prediction. We thus adopt the method of *ensemble learning* where multiple weak models are combined to develop a strong prediction model. In particular, we train our models on all the sampled datasets above, perform 10-fold cross validations, and then collect the predictions for *high* class returned by each of the models. Then we average such predictions of each candidate query for a change request, and identify such candidate that has the highest prediction for *high* quality (i.e., Lines 29–30, Algorithm 1, Lines 1–15, Algorithm 3, Steps 8-10, Fig. 3.2). Such candidate is then suggested as the search query from the change request for concept location by our technique.

### 3.3.9 A Working Example

Table 3.2 shows a working example of our technique—STRICT—for the showcase change request in Table 3.1. We employ three term weighting methods—TextRank, POSRank and WK-Core—and develop six candidate queries. It should be noted that each candidate query performs better than the baseline query (i.e., pre-processed *Title* from the request) which justifies our term weighting methods. However, we conduct further analysis on quality aspect of the queries, employ machine learning, and suggest the best performing candidate as the suggested query. For example, the candidate query— $CQ_{TR+POSR}$ —that uses both TextRank and POSRank for term weighting, performs the best. It returns the first correct result at the *third* position of the result list. Our approach identifies this candidate using machine learning, and suggests the query to the developer for code search. On the contrary, the baseline query returns such result at the 559<sup>th</sup> position which far below in the result list. Thus, our approach offers a major rank improvement over the baseline. Furthermore, tuned version of our suggested query—*{element IResource Provider Level Tree}*—returns the same result at the topmost position of the result list which is highly promising.

## 3.4 Experiment

Given two evaluation methods—*pre-retrieval* and *post-retrieval*—in the literature [93, 95], we choose post-retrieval method for the evaluation and validation of our search queries. This method evaluates the results retrieved by a query rather than simply analysing the query properties which makes it more reliable. Besides, past relevant studies [49, 98, 164] also adopted this method for evaluation and validation. We evaluate our

**Table 3.2:** A Working Example of Query Suggestion by STRICT

Technique	Search Query	QE
Baseline	$\mathbf{CQ}_T = \{Bug\ search\ Custom\ search\ hierarchically\ java\ search\ view\}$	559
TextRank (TR)	$\mathbf{CQ}_{TR} = \{search\ element\ check\ IJavaElement\ IJava\ Element\ Resource\ IResource\ Level\ Tree\}$	04
POSRank (POSR)	$\mathbf{CQ}_{POSR} = \{Resource\ element\ Java\ IResource\ superclasses\ Element\ Provider\ Bug\ search\ Custom\}$	77
<b>TR + POSR</b>	$\mathbf{CQ}_{TR+POSR} = \{element\ Resource\ IResource\ java\ IJavaElement\ IJava\ Element\ search\ Level\ Tree\}$	<b>03</b>
WK-CoreCOC	$\mathbf{CQ}_{COC} = \{search\ element\ IJavaElement\ IJava\ Element\ check\ java\ Resource\ IResource\ IAdaptable\}$	19
WK-CorePOS	$\mathbf{CQ}_{POS} = \{check\ Resource\ Element\ adapt\ IResource\ Java\ propose\ fail\ IJavaElement\ IJava\}$	77
WK-CoreCOC + WK-CorePOS	$\mathbf{CQ}_{COC+POS} = \{check\ Resource\ Element\ IResource\ IJavaElement\ IJava\ Element\ element\ search\ java\}$	89
$\mathbf{Q}_{BEST} = \text{getBestCandidateQuery}(QDM, \mathbf{CQ})$		
<b>STRICT</b>	$^*\mathbf{Q}_{BEST} = \{element\ Resource\ IResource\ java\ IJavaElement\ IJava\ Element\ search\ Level\ Tree\}$	<b>03</b>

\* = Suggested search query by our technique

technique using 2,885 software change requests with four state-of-the-art performance metrics. Furthermore, we compare with two state-of-the-art techniques on search query suggestion [120, 213] and conduct a user study involving five participants. Thus, we answer seven research questions as follows:

- **RQ<sub>1</sub>:** Can our suggested queries outperform the baseline queries from the change requests?
- **RQ<sub>2</sub>:** How do our suggested queries perform in retrieving the relevant source code documents?
- **RQ<sub>3</sub>:** How effective are the proposed term weighting algorithms –*TextRank*, *POSRank* and *WK-Core*– in identifying good quality search terms from a change request? How do they perform in comparison to the traditional term weighting methods (e.g., TF, TF-IDF)?
- **RQ<sub>4</sub>:** Are the parameters and thresholds adopted by our proposed technique justified?
- **RQ<sub>5</sub>:** Can STRICT outperform the state-of-the-art techniques in identifying good quality search terms from a change request?
- **RQ<sub>6</sub>:** Can STRICT outperform the state-of-the-art techniques in retrieving relevant source code documents from the corpus?
- **RQ<sub>7</sub>:** How does our working prototype perform in terms of usability and usefulness?

**Table 3.3:** Experimental Dataset

System	SLOC	#MD	#CR	Description
<code>eclipse.jdt.core-4.7.0</code>	951K	64K	404	Java infrastructure and compiler of Eclipse IDE
<code>eclipse.jdt.debug-4.6.0</code>	233K	16K	229	Debugging support module of Eclipse IDE
<code>eclipse.jdt.ui-4.7.0</code>	625K	57K	695	User Interface module of Eclipse IDE
<code>eclipse.pde.ui-4.7.0</code>	386K	32K	525	User Interface module for Eclipse IDE plug-ins.
<code>ecf-170.170</code>	222K	21K	345	An Eclipse communication framework
<code>log4j-1.2.17</code>	32K	3K	60	Apache logging service
<code>sling-0.1.10</code>	326K	30K	76	Apache framework for RESTful web applications
<code>tomcat-7.0.70</code>	286K	24K	551	Apache web server for Servlet and JSP
<b>Total #CR: 2,885</b>				

**SLOC:** Source Line Of Code, **MD:** Method Definitions, **CR:** Change Requests selected for experiments

### 3.4.1 Experimental Dataset

**Data Collection:** We collect a total of 2,885 software change requests from eight Java-based subject systems (i.e., five *Eclipse* systems + three *Apache* systems) for our experiments. Table 3.3 shows the details of our selected systems. We first collect the *RESOLVED* change requests from *BugZilla* and *JIRA* bug repositories of these systems, and then map them to their corresponding bug-fixing commits at GitHub and SVN. We analyse commit messages from each project, and identify specific Issue ID (i.e., identifier of a change task) in these messages using appropriate regular expressions [43]. Then, we include any change request in the experimental dataset only if there exists a corresponding commit pointing to the request. Such data acquisition approach is regularly adopted by the relevant literature [49, 98, 120, 276], and we also follow the same. In order to ensure a fair evaluation, we also discard the change requests from our dataset for which no source code files (i.e., Java classes) were changed or the relevant code files were missing from the system.

**Ground Truth Construction:** We collect the *changeset* (i.e., list of changed files) from each of our selected commits from the version control history, and develop solution set (i.e., *ground truth*) for the corresponding change tasks. Thus, for experiments, we collect not only the actual change requests from the reputed subject systems but also their solutions which were applied in practice by the developers [97]. We use several utility commands such as *git*, *clone*, *rev-list*, *shortlog* and *log* on GitHub and SVN consoles for collecting these information.

**Replication Package:** Our detailed experimental data, supporting materials and the working prototype are hosted online (<https://goo.gl/7Hrgmb>) for replication or third-part reuse.

### 3.4.2 Search Engine

We use a Vector Space Model (VSM) based search engine (e.g., *Apache Lucene* [95, 98]) to search for the documents that were changed in order to address the requested changes. Search engines generally index

the documents of a corpus prior to search. Lucene is mostly targeted for simple text documents (e.g., news article). Since source code documents in our projects contain items beyond regular texts (e.g., code segments), we apply limited preprocessing on them. In particular, we extract method bodies from each of the Java classes, and treat each method as an individual document in the corpus (Table 3.3). We remove all programming keywords, stop words and punctuation characters, and split the complex and camel case tokens into simpler ones [75]. For token splitting, we employ a state-of-the-art token splitting tool—*Samurai* [79]. Please note that we avoid stemming of the tokens for aforementioned reasons as described in Section 3.3.2 [120]. Once a search is initiated using a query, the search engine collects relevant documents from the corpus. It first uses a *Boolean Search Model* for short listing the documents, and then employs a *TF-IDF* based scoring technique (e.g., BM25 [227]) to return a ranked list of relevant documents. As existing studies suggest [120, 125], we consider the Top-10 results from the search engine for calculating the performance of our suggested queries.

One can argue about our choice of *Lucene* over the others (e.g., Indri [220]) as the back end code search engine. However, a recent third-party investigation<sup>2</sup> suggests that Apache Lucene and its variants (e.g., Apache Solr, Elasticsearch) have  $\approx 77\%$  market share in the enterprise search. Thus, Lucene is widely used for search operation not only in academia [98, 163, 164, 175, 176] but also in the industry level applications. Despite this strong supporting evidence, we also conduct an experiment to compare between *Lucene* and a potential contender – *Indri*. Our experiment suggests that Lucene can deliver 12% higher accuracy (Hit@10) than that of Indri for the same set of queries. More detailed supporting evidence could be found in Section 5.3.3, RQ<sub>1</sub>-(b) and Table 5.5. Thus, our choice of using Lucene is likely to be justified.

### 3.4.3 Performance Metrics

We choose four state-of-the-art performance metrics for evaluation and validation of our suggested queries. These metrics are frequently used by the relevant studies from the literature [98, 163, 164, 220], and thus are highly appropriate for our experiments as well.

**Query Effectiveness (QE):** It approximates a developer’s effort in locating the concept of interest in the source code [98, 163]. In practice, the metric returns the rank of the first correct result that matches with the ground truth, in the ranked list. The underlying idea is to provide an accurate starting point to the developer that deals with the concepts discussed in the change request. The lower the effectiveness value is, the better a query is since the developer can locate the correct result more quickly and with less efforts.

**Mean Reciprocal Rank (MRR@K):** Reciprocal rank@K refers to the multiplicative inverse of the rank of the first correctly returned result (i.e., matches with the ground truth) within the Top-K results [220, 276]. Mean Reciprocal Rank@K (MRR@K) averages such measures for all search queries ( $Q$ ) in the dataset. It can be defined as follows:

---

<sup>2</sup><https://www.datanyze.com/market-share/enterprise-search>

$$\text{MRR}(Q) = \frac{1}{|Q|} \sum_{q \in Q} \frac{1}{\text{firstRank}(q)}$$

Here,  $\text{firstRank}(q)$  provides the rank of the first correctly returned result. MRR can take a maximum value of 1.00. The bigger the MRR value is, the better the search query is.

**Mean Average Precision@K (MAP@K):** Precision@K refers to the precision calculated at the occurrence of every single relevant result in the ranked list [220, 276]. Average Precision@K (AP@K) averages the precision@K for all the relevant results in the list for a search query. Thus, Mean Average Precision@K is calculated from the mean of average precision@K for all the queries  $Q$  as follows:

$$\text{AP@K} = \frac{\sum_{k=1}^D P_k \times \text{rel}(k)}{|S|}, \quad \text{MAP@K} = \frac{\sum_{q \in Q} \text{AP@K}(q)}{|Q|}$$

Here,  $\text{rel}(k)$  denotes the relevance function of  $k^{\text{th}}$  result in the ranked list that returns either 0 or 1,  $P_k$  denotes the precision calculated at the  $k^{\text{th}}$  result, and  $D$  refers to the number of total results.  $S$  is the true positive retrieved by a query, and  $Q$  is the set of all queries (i.e., change requests).

**Top-K Accuracy / Hit@K:** It refers to the percentage of the search queries (i.e., change requests) for each of which at least one result file is correctly returned (i.e., matches with the ground truth) within the Top-K result list [239, 250, 276].

### 3.4.4 Evaluation of STRICT

We conduct experiments using 2,885 change requests from eight subject systems (Table 3.3) where the above four metrics (Section 3.4.3) are applied to performance evaluation. We run each of our suggested queries with Lucene search engine (Section 3.4.2), check their results against the ground truth (Section 3.4.1), and then compare with the baseline queries from those requests. In this section, we discuss our evaluation details and answer **RQ<sub>1</sub>–RQ<sub>4</sub>** as follows.

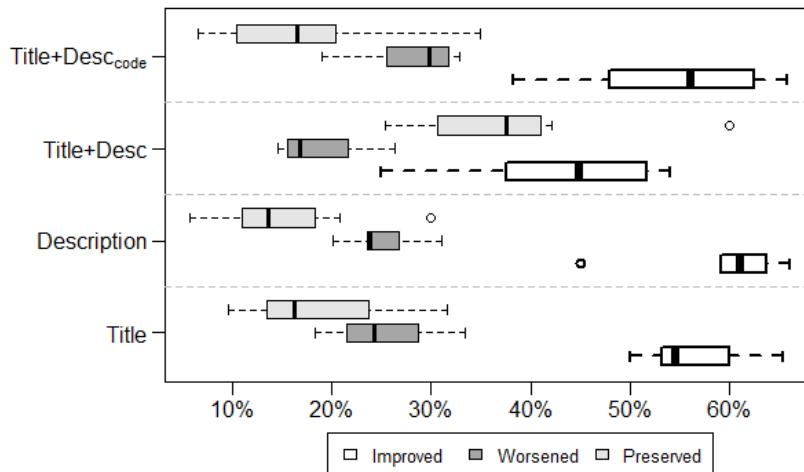
**Baseline Query Selection:** Developers often copy and paste the texts from a software change request on an ad-hoc basis, and search for the source code that needs to be changed. Hence, request texts can be considered as the baseline queries for our evaluation or validation [49, 98, 120]. We consider four types of baseline queries from each request using its fields and structures. In particular, we capture *Title* and *Description* fields from each change request, conduct standard natural language preprocessing (i.e., removal of stop words and punctuation marks, splitting of complex or camel case tokens) on them, and then prepare three baseline queries– Title, Description, Title + Description. It should be noted that the same set of preprocessing steps were also applied in our approach in Section 3.3.2. Structured tokens (e.g., camel-case structures) are reported as better keywords than unstructured natural language terms by several earlier studies [49, 120, 191]. We thus extract the structured tokens from each request using appropriate regular expressions. Then we split these tokens using a state-of-the-art token splitting tool–*Samurai* [79], and then prepare another baseline query, (Title+Description)<sub>code</sub>, for our experiments.

**Table 3.4:** Comparison of Query Effectiveness between STRICT and Baseline Queries

Query Pairs	#Queries	Improved/(MRD)	Worsened/(MRD)	p-value/ $\Delta$	Preserved	Net MRD
STRICT vs. Title	2,885	56.41%/(-198)	25.06%/(+175)	*0.02/1.00 (large)	18.53%	-23
STRICT vs. Description	2,885	<b>59.91%</b> /(-406)	24.94%/(+200)	*0.008/1.00 (large)	15.16%	-206
STRICT vs. Title + Description	2,885	43.41%/(-258)	18.59%/(+206)	*0.008/0.97 (large)	38.01%	-52
<b>STRICT vs. (Title + Description)<sub>code</sub></b>	<b>2,885</b>	<b>54.71%</b> /(-402)	<b>28.26%</b> /(+199)	<b>*0.008/1.00 (large)</b>	<b>17.03%</b>	<b>-203</b>
<b>Comparison with Difficult Baseline Queries</b>						
STRICT vs. Title	1,898	<b>69.06%</b> /(-248)	22.43% (+262)	*0.008/1.00(large)	8.52%	+14
STRICT vs. Description	2,035	<b>74.34%</b> /(-474)	19.90%/(+312)	*0.008/1.00(large)	5.76%	-162
STRICT vs. Title + Description	1,958	56.10%/(-305)	16.84%/(+295)	*0.008/1.00(large)	27.06%	-10
<b>STRICT vs. (Title + Description)<sub>code</sub></b>	<b>1,858</b>	<b>71.51%</b> /(-486)	<b>21.93%</b> /(+360)	<b>*0.008/1.00(large)</b>	<b>6.57%</b>	<b>-126</b>

\* = Significant difference between improvement and worsening, MRD = Mean Rank Difference between STRICT and baseline queries,  $\Delta$  = Cliff's Delta, effect size of significance





**Figure 3.3:** Improvement, worsening and preserving of the baseline queries by our proposed technique – STRICT

**Query Improvement, Worsening and Preserving:** We use these terminologies frequently throughout the rest of the discussions on experiments, and they are defined as follows. If a suggested query provides a better rank than its corresponding baseline query for the first correct result, then we call it *query improvement* and vice versa as *query worsening*. On the contrary, if both queries provide the same rank, then we call it *query preserving* by the suggested query. Thus, a technique that improves more baseline queries than it worsens is a better technique than the baseline approaches.

**Answering RQ<sub>1</sub>–Comparison with Baseline Queries:** We execute each of the baseline queries with our search engine–*Lucene*, and collect the ranks of their first correct results in the list. We also collect similar ranks returned by our queries suggested from each of the change requests, and compare with that of the baseline queries. Tables 3.4, 3.5 and Fig. 3.3 summarize our comparative analysis. From Table 3.4 (upper half), we see that our queries provide higher ranks than 43%–60% of the baseline queries. Such statistic is promising according to the relevant literature that reported a maximum of 55% improvement on a different dataset [98, 164]. Our queries also provide relatively lower ranks than 19%–28% of the baseline queries on average. Given these mixed findings, we compare the query improvement ratios of our technique against the worsening ratios across eight subject systems using statistical tests (e.g., *Wilcoxon Signed Rank*, *Cliff’s Delta*). As shown in Table 3.4, the query improvements by our technique are significantly higher (i.e.,  $p\text{-value} < 0.05$ ) than the worsening ratios with large effect sizes (i.e.,  $0.97 \leq \Delta \leq 1.00$ ). Mean Rank Difference (MRD) calculates the rank difference between baseline and suggested queries where a *negative value* refers to *rank improvement* and vice versa as *rank worsening*. According to MRD analysis, our queries push the first correct results towards the top of the list by 23 to 206 positions which is a major rank improvement over the baseline counterpart.

We also consider Effectiveness of the baseline queries and divide them into two categories–*easy* and *difficult*. If a baseline query returns its first correct result within the Top-10 positions of the list, we call it *easy* query and vice versa as *difficult* query [96, 98]. We found that the difficult baseline queries are

**Table 3-5:** Effectiveness Details of STRICT Query vs. Baseline Queries, (Title+Description)<sub>code</sub>

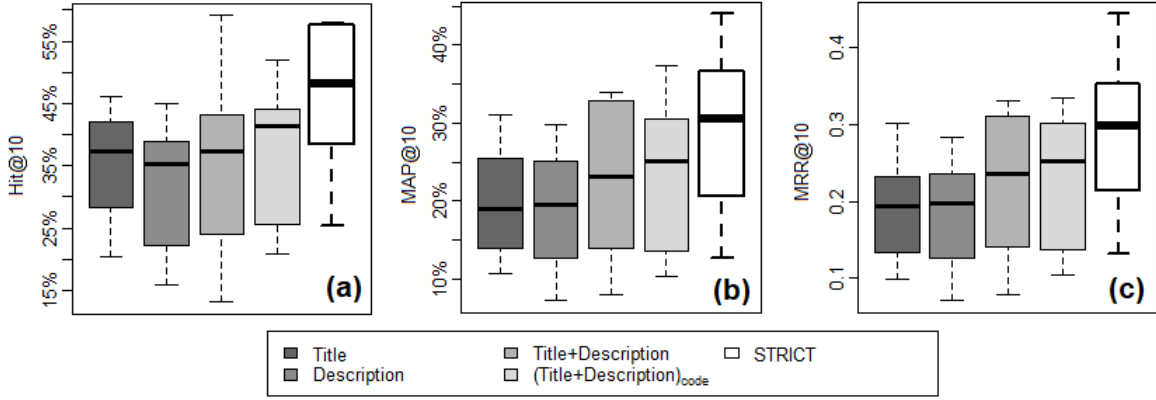
System	#Q	Improvement							Worsening							Preserving	
		#Improved	Mean	Q1	Q2	Q3	Min.	Max.	#Worsened	Mean	Q1	Q2	Q3	Min.	Max.	#Preserved	
eef	345	<b>184</b> Base Rank→	95	<b>2</b>	<b>10</b>	39	1	3,182	106	223	12	45	146	2	2,345	<b>55</b>	
core	404	<b>248</b> Base Rank→	157	<b>1</b>	<b>5</b>	44	1	6,422	77	260	15	60	172	2	5,171	<b>79</b>	
debug	229	<b>146</b> Base Rank→	613	11	64	384	2	9,523	Base Rank→	36	2	13	50	1	454	-	
jdt.ui	695	<b>458</b> Base Rank→	149	<b>9</b>	29	79	1	3,888	68	346	31	75	395	4	2,598	<b>15</b>	
pde.ui	525	<b>310</b> Base Rank→	382	42	107	323	2	5,051	Base Rank→	187	5	24	176	1	2,953	-	
log4j	60	<b>23</b> Base Rank→	174	<b>2</b>	12	59	1	9,872	169	496	12	55	311	2	9,569	<b>68</b>	
sling	76	<b>38</b> Base Rank→	746	27	139	697	2	9,373	Base Rank→	237	3	12	94	1	4,801	-	
tomcat70	551	<b>310</b> Base Rank→	190	<b>5</b>	18	91	1	4,697	157	515	19	68	393	2	9,701	<b>58</b>	
		<b>23</b> Base Rank→	542	35	150	544	2	8,919	Base Rank→	216	4	16	85	1	4,292	-	
		<b>23</b> Base Rank→	57	<b>3</b>	<b>5</b>	44	1	786	16	46	<b>7</b>	13	53	2	224	<b>21</b>	
		<b>38</b> Base Rank→	63	12	42	80	2	370	Base Rank→	26	2	4	20	1	185	-	
		<b>38</b> Base Rank→	119	<b>1</b>	<b>5</b>	20	1	2,483	25	249	<b>10</b>	39	225	2	1,547	<b>13</b>	
		<b>253</b> Base Rank→	602	12	102	496	2	3,830	Base Rank→	71	1	6	45	1	896	-	
		<b>253</b> Base Rank→	55	<b>2</b>	<b>7</b>	38	1	1,147	181	132	<b>6</b>	21	99	2	2,406	<b>117</b>	
		<b>253</b> Base Rank→	201	9	41	182	2	4,135	Base Rank→	45	1	6	23	1	992	-	
<b>Total:</b>	<b>2,885</b>	<b>1,660</b>							799							<b>426</b>	

core = eclipse.jdt.core, debug = eclipse.jdt.debug, jdt.ui = eclipse.jdt.ui, pde.ui = eclipse.pde.ui, Mean=Mean rank of first relevant document in the search result,  $Q_i$ = Rank value for  $i^{th}$  quartile of all result ranks

**Table 3.6:** Document Retrieval Performance of STRICT Queries

Query	#Keywords	Hit@1	Hit@5	Hit@10	MAP@10	MRR@10
Title	08	13.24%	26.76%	35.23%	19.78%	0.19
Description	72	13.03%	25.52%	31.70%	18.91%	0.18
Title+Description	100	17.32%	29.36%	35.21%	22.70%	0.22
<b>(Title+Description)<sub>code</sub></b>	<b>34</b>	<b>16.89%</b>	<b>30.59%</b>	<b>36.88%</b>	<b>23.21%</b>	<b>0.23</b>
STRICT <sub>T10</sub>	10	13.18%	26.60%	32.33%	18.95%	0.19
STRICT <sub>T20</sub>	20	17.84%	33.18%	40.65%	24.82%	0.24
STRICT <sub>T30</sub>	30	19.31%	35.69%	43.90%	26.91%	0.26
<b>STRICT</b>	<b>37</b>	<b>*21.95%</b>	<b>*38.07%</b>	<b>*46.43%</b>	<b>*28.99%</b>	<b>*0.29</b>

**Emboldened** = Comparatively higher, \* = Statistically significant difference between suggested and the baseline queries

**Figure 3.4:** Comparison of the document retrieval performance of STRICT queries against baseline queries in terms of (a) Hit@10, (b) MAP@10, and (c) MRR@10

significantly improved by our approach. From Table 3.4 (lower half), we see that our approach improve 56%–74% of the difficult baseline queries while worsening only 17%–22%. Thus, our query improvements are three to four times higher than the query worsening ratios which is highly promising. We also found that the easy queries cannot be further improved by our approach rather their quality level remain unchanged. Such finding is also consistent with the earlier findings from the literature [98, 189].

Earlier studies report significant benefit in including source code tokens in the search query for bug localization [163] and feature location [49]. We thus compare with another baseline,  $(Title+Description)_{code}$  that comprises of only structured or code-like tokens (e.g., camel-case tokens, `HashSet`) and their split versions extracted from the change request. Although these queries perform highly compared to other three baselines, our suggested queries perform even higher. From Table 3.4 (upper part and lower part), we see that STRICT can improve 55%–72% of the  $(Title+Description)_{code}$  queries in two different circumstances which demonstrates the high potential of our approach for the query suggestion.

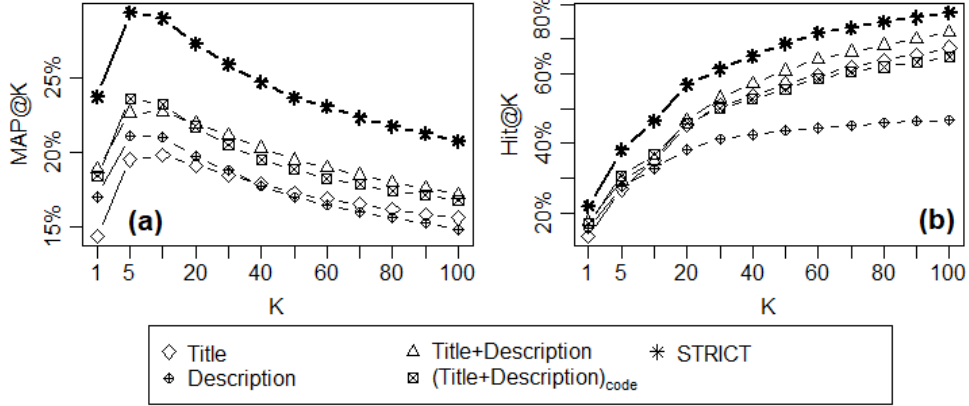
We also further investigate the ranks of first correct results returned by our approach, and compare them with the ranks of  $(Title+Description)_{code}$ , one of the high performing baselines. In particular, we compare

the rank distribution of STRICT with that of the baseline queries using descriptive statistics. From Table 3.5, we see that the rank improvements by our technique are generally higher than the corresponding rank worsening. For example, the mean baseline rank of 248 queries from `eclipse.jdt.core` system is 613. Our technique improves that rank to 157, and returns correct results for at least 50% of these queries within the Top-5 positions of the result list (i.e.,  $Q_2 = 5$ ) which are promising. Our technique also worsens the mean rank of 77 baseline queries of the same subject system from 36 to 260. We compare such rank improvement or worsening of STRICT against the baseline across eight subject systems, and found statistical significance (i.e.,  $p\text{-value}=0.02<0.05$ ) with a medium effect size (i.e.,  $\Delta = 0.38$ ). In fact, our mean rank improvement (i.e., -402) is two times higher than the mean rank worsening (i.e., +199). According to our quantile analysis, STRICT turns 50% of the *difficult* baseline queries into *easy* ones (i.e., rank within Top-10 positions) from five systems – `ecf`, `eclipse.jdt.core`, `log4j`, `slings` and `tomcat70`. Fig. 3.3 further demonstrates the box plots of improvement and worsening of the baseline queries by our technique across eight subject systems. We see that the medians of query improvement ratios of our technique are much higher than that of the corresponding worsening ratios. In fact, STRICT improves 50%–66% of the queries for six out of eight systems which is promising.

We also manually investigate the baseline queries that are worsened by our approach, and found two important observations. First, the extent of our rank worsening is relatively smaller than that of rank improvement. Second, most of these worsened queries contain structured entities (e.g., stack traces). STRICT could have performed even higher if such structures were properly incorporated in the text graphs. Such an issue has actually been resolved by our later study called BLIZZARD (Chapter 5). We also manually investigate the source code tokens in the baseline queries. We found that such items can always not be guaranteed in the request texts. According to an earlier study [248], up to 55% of the change requests of a system might contain only unstructured plain texts. Thus, the performance of the code-only baseline queries,  $(\text{Title}+\text{Description})_{\text{code}}$ , could be limited. On the contrary, we suggest relatively higher quality queries from the careful analysis of any available textual information in the change request. All these empirical evidences presented above demonstrate the high potential of our approach.

**Summary of RQ<sub>1</sub>:** STRICT improves **43%-74%** of the baseline queries. Furthermore, the extent of our result rank improvement is **two** to **three** times higher than that of the rank worsening.

**Answering RQ<sub>2</sub> – Document Retrieval Performance:** Although our approach improves majority of the baseline queries in RQ<sub>1</sub>, we further evaluate the approach in terms of document retrieval performance. While RQ<sub>1</sub> considers only first relevant document of the result list, we consider all the relevant documents retrieved by a query in RQ<sub>2</sub>. We execute each of our queries, analyse the Top-10 results (as many existing studies do [98, 120, 239]), and calculate Hit@K, mean average precision@K and mean reciprocal rank@K of our technique. Table 3.6, Figures 3.4 and 3.5 summarize our performance details. From Table 3.6, we see that our queries return correct results for 46% of 2,885 change requests with 29% mean average precision@10 and a mean reciprocal rank@10 of 0.29 which are 26%, 25% and 26% higher respectively than the best baseline



**Figure 3.5:** Comparison of STRICT queries with baseline queries for Top 1 to 100 results in terms of (a) MAP@K and (b) Hit@K

measures (i.e.,  $(Title+Description)_{code}$ ). Such performances are also comparable to other earlier findings that use different datasets [37, 49, 59], which signals the external validity of our results. Our queries also have more potential for practical use than the baseline queries. We achieve relatively higher performance using a limited number of the search terms. For example, two of the baseline approaches—*Description* and *Title+Description*—achieve 32%–35% Hit@10 with 72–100 keywords in each of their queries on average. Such long queries are noisy, and difficult to tweak manually. On the other hand, we offer 38% Hit@10 using only top 10 of our suggested keywords.  $(Title+Description)_{code}$  performs the best among the baseline approaches, and it has a query length comparable to ours (e.g., 34). We thus consider it as an appropriate baseline opponent and compare with it more extensively using statistical tests. Our tests report that the performance of our query is significantly higher (i.e.,  $p\text{-value} < 0.05$ ) with a medium effect size (i.e.,  $\Delta = 0.34$ ) in terms of Hit@K, MAP@10 and MRR@10.

Fig. 3.4 further contrasts our queries against the baseline in terms of (a) MAP@10, (b) MRR@10 and (c) Hit@10 using box plots over all the subject systems. We see that our technique, STRICT, has a higher median (i.e., 50% percentile) for each of the three performance measures. That is, top 50% of our measures are comparatively higher than the top 50% of any baseline measures. In other words, our suggested queries can retrieve the relevant source code documents more efficiently than the baseline ones.

Although our queries show high performance for Top-10 results, we further investigate how they perform when more results (e.g., 100) are considered. Fig. 3.5 shows (a) MAP@K and (b) Hit@K for Top-1 to Top-100 results. We see that STRICT achieves a Top-50 accuracy of 69% and Top-100 accuracy of 78% which are 24% and 20% higher respectively than the equivalent baseline measures,  $(Title+Description)_{code}$ . More importantly, our accuracy measures remained significantly higher (i.e.,  $p\text{-value} < 0.05$ ) than the corresponding baseline measures with *small to large* effect sizes (i.e.,  $0.24 \leq \Delta \leq 0.64$ ) for various Top-K results. Our queries also remained significantly more precise than the baseline queries (i.e.,  $p\text{-value} < 0.05$ ,  $0.42 \leq \Delta \leq 0.96$ ) across all Top-K results which demonstrates the relative strength of our proposed approach over the baseline.

**Summary of RQ<sub>2</sub>:** Our queries achieve **26%** higher accuracy, **25%** higher precision and **26%** higher reciprocal rank than the best performing baseline when Top-10 results are analysed. Furthermore, STRICT is found *more accurate* and *more precise* than the baseline when Top-100 results are analysed.

**Answering RQ<sub>3</sub> – (a) Role of our Term Weighting Algorithms:** We investigate how three term-weighting algorithms – *TextRank*, *POSRank* and *Weighted K-Core* – perform in identifying good quality search terms from a software change request. Table 3.7 and Fig. 3.6 summarize our investigation details. From Table 3.7, we see that both TextRank and POSRank perform almost equally in terms of Hit@K, MAP@10 and MRR@10. Their combination marginally improves upon the individual performances. On the contrary, WK-Core is a relatively better approach for term weighting. It achieves 41% Hit@10 as opposed to 39% of TextRank and POSRank. Our technique, STRICT, combines all three term-weighting algorithms using machine learning, and achieves 46% Hit@10 which is  $\approx 16\%$  higher. Similar findings are also observed in the case of precision and reciprocal rank. Fig. 3.6 further demonstrates the improvement, worsening and preserving ratios of the baseline queries by each of the term weighting approaches. We see that the combination of all three approaches benefits the baseline queries on average. For example, TextRank improves 48% and worsens 39% of the baseline queries. On the contrary, our technique that combines all three term weighting approaches improves 55% and worsens 28% of the queries which are 14% higher and 27% lower respectively. Thus, our choice of combining different term weighting algorithms is justified.

**(b) Comparison with Traditional Term Weighting Algorithms:** TF-IDF has been a popular term weighting approach for over the last five decades [114]. It stands for term frequency (TF) times the inverse document frequency (IDF), i.e.,  $TF\text{-}IDF = TF \times IDF$ . While TF counts the occurrences of a term within a document, IDF is computed using the number of documents from corpus containing that term. Thus, TF-IDF captures both local and global contributions of a term, and determines its importance. We compare our term weighting approaches with this traditional approach, and demonstrate the potential of our approaches. From Table 3.8, we see that IDF performs the best among three traditional approaches - TF, IDF and TF-IDF. It improves 1,467 baseline queries and retains the rank of 281 queries. Thus, the traditional approaches improve or at least preserve 1,748 (60.59%) of 2,885 baseline queries. On the contrary, our combined term weight, {TextRank+POSRank+WK-Core}, improves or preserves 1,784 (61.84%) baseline queries which is comparatively higher. Furthermore, when our term weighting approaches are combined with machine learning, they improve or preserve 70% of the baseline queries which is 16% higher than that of traditional approaches.

**Summary of RQ<sub>3</sub>:** Each of the term weighting algorithms has its own strengths and weaknesses. Our approach achieves **16%** higher Hit@10, improves **15%** more of the baseline queries, and worsens **24%** less of the queries when all three algorithms are combined using machine learning. Furthermore, our term weighting algorithms are more promising than the traditional counterparts.

**Table 3.7:** Retrieval Performance of TextRank, POSRank and WK-Core

Term weight	Hit@1	Hit@5	Hit@10	MAP	MRR
TextRank	14.75%	30.30%	39.47%	22.20%	0.22
POSRank	14.68%	29.67%	38.97%	22.29%	0.21
<b>{TextRank + POSRank}</b>	<b>16.55%</b>	<b>31.29%</b>	<b>39.94%</b>	<b>23.51%</b>	<b>0.23</b>
WK-Core <sub>COC</sub>	16.57%	30.73%	40.58%	23.14%	0.23
WK-Core <sub>POS</sub>	15.80%	30.49%	40.80%	23.63%	0.23
{WK-Core <sub>COC</sub> + WK-Core <sub>POS</sub> }	16.74%	31.67%	41.03%	24.19%	0.23
<b>STRICT</b>	<b>*21.95%</b>	<b>*38.07%</b>	<b>*46.43%</b>	<b>*28.99%</b>	<b>*0.29</b>

**Table 3.8:** Comparison between Proposed and Traditional Term Weights

Term weight	#Queries	Improved	Worsened	Preserved
TF	2,885	1,439 (49.88%)	1,112 (38.54%)	247 (8.56%)
IDF	2,885	1,467 (50.85%)	1,031 (35.74%)	281 (9.74%)
TF-IDF	2,885	1,439 (49.88%)	1,082 (37.50%)	267 (9.25%)
TextRank	2,885	1,470 (50.95%)	1,077 (37.33%)	256 (8.87%)
POSRank	2,885	1,397 (48.42%)	1,124 (38.96%)	274 (9.50%)
{TextRank + POSRank}	2,885	1,445 (50.09%)	1,056 (36.60%)	304 (10.54%)
<b>{TextRank + POSRank + WK-Core}</b>	<b>2,885</b>	<b>1,476 (51.16%)</b>	<b>1,021 (35.39%)</b>	<b>308 (10.68%)</b>
<b>STRICT</b>	<b>2,885</b>	<b>1,660 (57.54%)</b>	<b>792 (27.45%)</b>	<b>366 (12.69%)</b>

**Answering RQ<sub>4</sub> – Impact of the Adopted Parameters and Thresholds:** We conduct experiments to justify our choice on the suggested query length, the use of data re-sampling, and the use of machine learning algorithm. Figure 3.7 demonstrates the impacts of various parameters and thresholds upon the performance of our approach. From Fig. 3.7-(a), we see that our approach improves more baseline queries than it actually worsens when only Top-10 terms from each of our queries are used for code search. This performance gradually improves up to a query length of  $\approx 35$  which makes it a potential threshold. Similar findings can also be observed in Fig. 3.7-(b). However, earlier studies [50, 153, 217] report significant benefits of a dynamic threshold over a fixed threshold. Furthermore, about 50% of our change requests have a query length of  $\approx 40$ . Thus, almost all terms from these change requests are likely to return as search terms with a fixed threshold of 35 which makes the automated query suggestion irrelevant. Hence, we use a dynamic threshold for our query length rather than a fixed threshold. In particular, we choose the top 33% of the highly weighted terms from each change request as our query. Earlier studies [50, 153, 217] based on graph-based term weighting use such threshold for keyword selection. Such threshold also helps us achieve the optimal performance (i.e., dashed and dotted lines) both in query improvement and in the retrieval of relevant source documents. More interestingly, such threshold ensures an average length of 37 for our suggested queries which is also close to 35.

**Table 3.9:** Comparison of Baseline Query Improvements between STRICT and Existing Techniques

Query Pairs	#Queries	Improved/(MRD)	Worsened/(MRD)	Preserved	Net Gain	Net MRD
Kevic and Fritz [120]-I	2,885	38.52%/-505)	49.82%/+288)	11.67%	-11.30%	-
Kevic and Fritz [120]-II	2,885	47.28%/-455)	40.16%/+228)	12.55%	+7.12%	-227
Rocchio [213]-I	2,885	43.53%/-479)	47.96%/+395)	8.51%	-4.43%	-
Rocchio [213]-II	2,885	45.41%/-436)	41.73%/+302)	12.86%	+3.68%	-134
Rahman and Roy [191]	2,885	47.59%/-424)	37.16%/+200)	15.25%	+10.43%	-224
<b>STRICT</b>	<b>2,885</b>	<b>54.71%/-402)</b>	<b>28.26%/+199)</b>	<b>17.03%</b>	<b>+26.45%</b>	<b>-203</b>
<b>Comparison using Difficult Baseline Queries</b>						
Kevic and Fritz [120]-I	1,858	53.02%/-578)	38.89%/+406)	8.09%	+14.13%	-172
Kevic and Fritz [120]-II	1,858	63.33%/-532)	30.10%/+387)	6.57%	+33.23%	-145
Rocchio [213]-I	1,858	59.76%/-553)	35.04%/+602)	5.20%	+24.72%	+49
Rocchio [213]-II	1,858	60.74%/-520)	33.58%/+494)	5.68%	+27.16%	-26
Rahman and Roy [191]	1,858	64.36%/-497)	28.43%/+361)	7.22%	+35.93%	-136
<b>STRICT</b>	<b>1,858</b>	<b>71.51%/-486)</b>	<b>21.93%/+360)</b>	<b>6.57%</b>	<b>+49.58%</b>	<b>-126</b>

We also investigate whether the re-sampling of training data has any impact upon our performance or not. In particular, we compare performance between two variants of our approach where one variant uses data re-sampling and the other does not. Our statistical tests report that re-sampling based variant achieves significantly higher accuracy (Hit@K) and significantly higher precision (MAP@K) with *small to large* effect sizes (i.e., all  $p\text{-values} \leq 0.05$ ,  $0.26 \leq \Delta \leq 0.68$ ).

We also investigate whether the use of any particular machine learning algorithm in our approach makes a difference or not. In particular, we compare performance between two variants of our approach where one variant use RandomForest and the other uses Classification and Regression Tree (CART) as the learning algorithm. Our statistical tests report that RandomForest-based variant achieves significantly higher accuracy (Hit@K) and significantly higher precision (MAP@K) than the CART-based variant with *small to medium* effect sizes (i.e., all  $p\text{-values} \leq 0.05$ ,  $0.17 \leq \Delta \leq 0.46$ ).

**Summary of RQ<sub>4</sub>:** We use a *dynamic threshold* for our query length, apply **re-sampling** to our training data to cater for data skewness, and employ **RandomForest** as our learning algorithm for search query suggestion. All these choices of ours are justified by the appropriate empirical evidences above.

### 3.4.5 Comparison with Existing Techniques

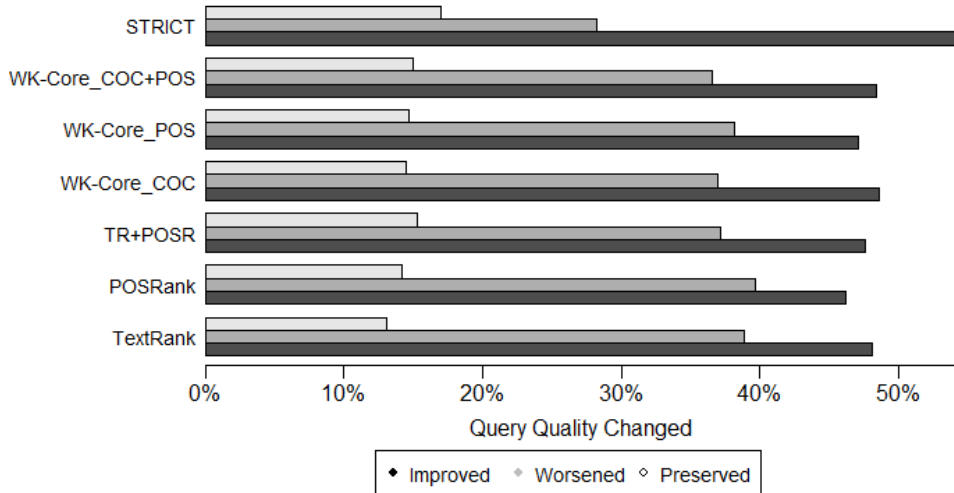
Although our suggested queries outperform the baseline queries with a large margin, we further compare our approach with the state-of-the-art. Our approach, STRICT, could be considered as a technique both for (1) search term identification and for (2) query reformulation. That is, STRICT not only identifies high quality search terms from a change request but also, in essence, reformulates the baseline query by removing the low quality search terms. We thus compare our approach with the state-of-the-art studies [120, 213] from these



**Table 3.10:** Comparison of Query Effectiveness with Existing Query Reformulation Techniques

Technique	Improvement						Worsening						Preserving		
	#IQ	Mean	Q1	Q2	Q3	Min.	Max.	#WQ	Mean	Q1	Q2	Q3	Min.	Max.	#PQ
Kevic and Fritz-I	1,179	135	4	21	85	1	8,406	1,218	361	20	78	310	2	6,545	192
Kevic and Fritz-II	1,436	114	3	15	61	1	7,609	1,112	341	14	61	262	2	8,280	236
Rocchio-I	1,303	219	3	22	117	1	9,335	1,370	493	22	93	411	2	9,260	145
Rocchio-II	1,341	234	4	24	111	1	7,758	1,230	429	19	71	315	2	9,547	264
Rahman and Roy	1,445	159	3	17	77	1	7,371	1,056	318	11	49	231	2	9,569	304
Baseline	-	505	18	88	393	2	9,523	-	137	2	10	56	1	4,801	-
<b>STRICT</b>	<b>1,660</b>	<b>142</b>	<b>2</b>	<b>12</b>	<b>58</b>	<b>1</b>	<b>9,872</b>	<b>792</b>	<b>329</b>	<b>12</b>	<b>47</b>	<b>214</b>	<b>2</b>	<b>9,701</b>	<b>366</b>

**IQ=Improved Baseline Queries, WQ=Worsened Baseline Queries, PQ=Preserved Baseline Queries**

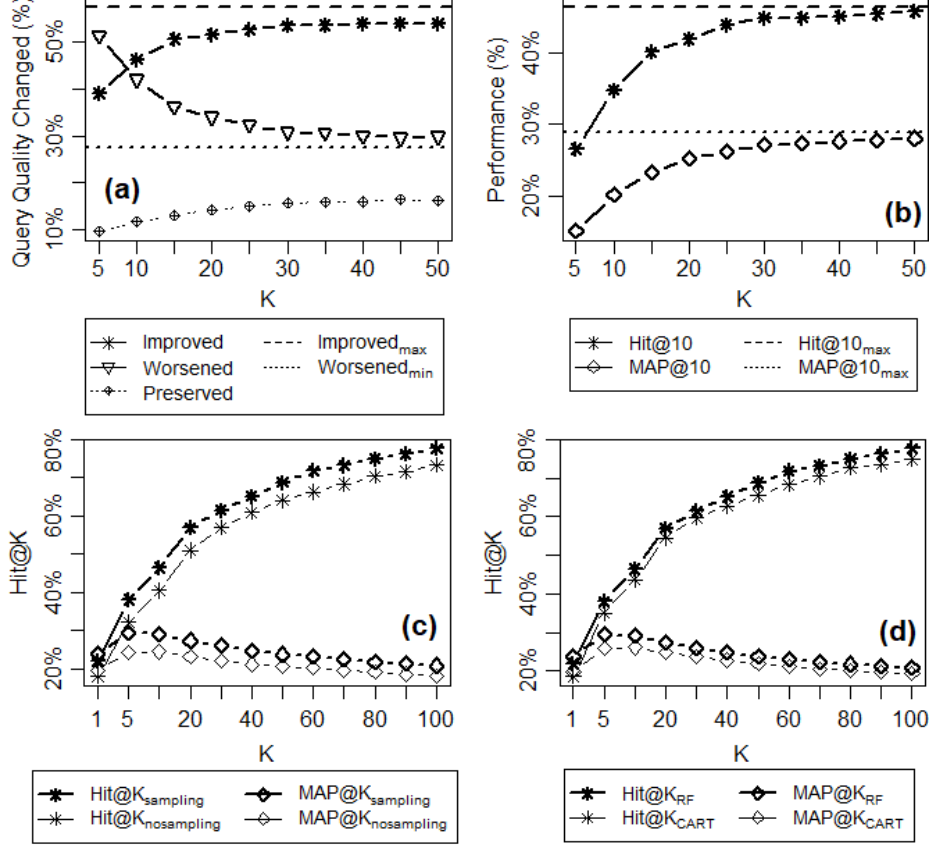


**Figure 3.6:** Role of three term weighting algorithms in the improvement, worsening and preserving of the baseline queries

two domains above. Kevic and Fritz [120] use a regression model to identify search terms from a change request. They employ several heuristics concerning *frequency*, *location*, *part of speech*, and *notation* of the terms from the request texts. To the best of our knowledge, Kevic and Fritz [120] is the only available study in the literature for search term identification from a change request which makes it the state-of-the-art. Rocchio [213] collects top  $K$  (e.g.,  $K = 5$ ) documents from the corpus returned by a given query, identifies appropriate candidate terms from these documents using term weighting (e.g., TF-IDF [114]), and then expands the query. Such expansion strategy has been adopted by a number of studies on query reformulation [84, 98, 168, 189, 251] which makes it a suitable candidate for our comparison. We implement both of these approaches using the authors’ provided settings and parameters (e.g., metric weights), and collect the search queries returned by them for our dataset. In particular, we develop two variants of each approach where partial and the whole texts of a request are used as their inputs. We also select our earlier work, Rahman and Roy [191], for comparison which is essentially the current state-of-the-art on search term identification. We compare our queries with the queries from these five existing techniques using two performance dimensions – (1) query rank improvement, and (2) relevant source document retrieval, and answer  $RQ_5$  and  $RQ_6$  as follows:

**Answering  $RQ_5$ –Comparison with the State-of-the-Art using Query Rank Improvement:**

Table 3.9 compares our approach with the existing approaches in terms of baseline query improvement, worsening and preserving ratios. We see that Kevic and Fritz-II performs the best among the existing approaches, and improves 47% of the baseline queries while worsening 40%. On the contrary, our approach improves 55% of the baseline queries and worsens only 28% of the queries which are 16% higher and 30% lower respectively. That is, while the existing works improve the baseline queries, they also degrade a significant amount of queries which makes the net gain (i.e., improvement ratio – worsening ratio) insignificant. For example, none of the existing approaches achieves a net gain over 10%. On the contrary, our approach

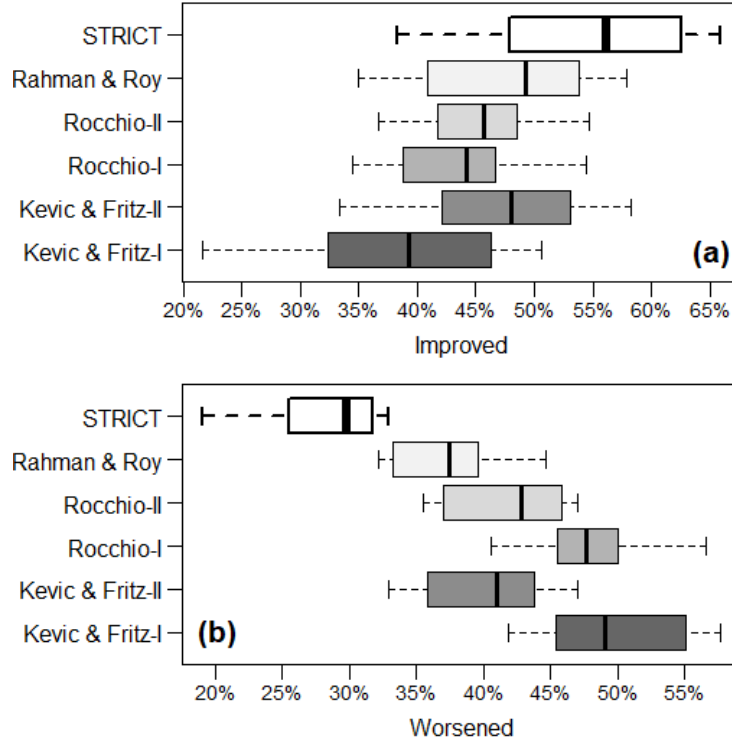


**Figure 3.7:** Impact of the adopted parameters and thresholds – (a,b) suggested query length, (c) use of data re-sampling, and (d) use of machine learning algorithm

achieves a net gain improvement of 27% which is highly promising. According to mean rank difference (MRD) analysis, our approach improves the result ranks up to 203 positions. This work also achieves 15% more improvement than our earlier work Rahman and Roy, and thus advances the state-of-the-art.

We also compare our approach against the existing approaches using the difficult baseline queries that return their results beyond 10<sup>th</sup> position of the result list. From Table 3.9 (lower part), we see that Kevic and Fritz-II improves 63% of the queries and worsens 30% of the queries. On the contrary, our approach improves 72% and worsens 22% which are 13% higher and 27% lower respectively. Thus, our approach outperforms the five existing techniques above in two different circumstances which clearly demonstrates its high potential for query suggestion.

We also analyse the distribution of result ranks, and compare our approach with the existing approaches using such analysis. From Table 3.10, we see that Kevic and Fritz-II and our earlier work are strong competitor of each other. They improve a maximum of 1,445 queries with 25% quantile at 3 and 50% quantile at 15. On the contrary, our approach improves a total of 1,660 baseline queries with 25% quantile at 2 and 50% quantile at 12 which are 15%, 33% and 20% higher respectively. While each of the existing approaches provide relatively better ranks than the baseline, our ranks are even more promising. Furthermore, our approach



**Figure 3.8:** Comparison of baseline query improvements or worsening between our technique, STRICT, and the existing techniques

worsens the least amount (i.e., 792) of baseline queries while ensuring the maximum improvement (i.e., 1,660) at the same time.

We also further analyse the query improvement and worsening ratios of eight subject systems, and compare our approach with the existing ones using box plots. From Fig. 3.8, we see that the median of improvement and worsening ratios of Kevic and Fritz-II are 48% and 41%. On the contrary, our approach achieves 56% improvement and 30% worsening which are 17% higher and 27% lower respectively. All these evidences above clearly demonstrate the superiority of our approach over the state-of-the-art.

**Summary of RQ<sub>5</sub>:** Our approach clearly outperforms the existing approaches including the state-of-the-art in improving the baseline queries. It improves **16%** more and worsens **30%** less of the baseline queries. Furthermore, it delivers **two to three** times higher *net gain* in the result ranks than that of the state-of-the-art approaches.

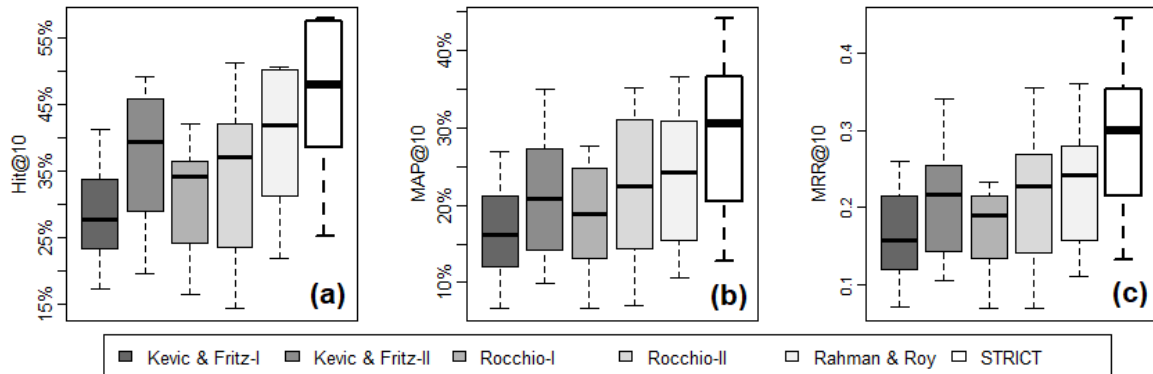
**Answering RQ<sub>6</sub>—Comparison with the State-of-the-Art using Relevant Document Retrieval:**

While our approach outperforms the existing approaches on query improvement, we further compare using relevant document retrieval performance. From Table 3.11, we see that Kevic and Fritz-II achieves 37% Hit@10 with 21% precision and a reciprocal rank of 0.21 when only Top-10 results are analysed. On the other hand, our approach achieves 46.43% Hit@10, 29% MAP and 0.29 reciprocal rank which are 25%, 37% and 38% higher respectively. Our performance measures are also 16%, 23% and 26% higher respectively than that of our earlier work [191] which demonstrate the potential of STRICT over the state-of-the-art. When

**Table 3.11:** Comparison with Existing Techniques in Document Retrieval

Technique	Hit@1	Hit@5	Hit@10	MAP	MRR
Kevic and Fritz-I	11.87%	22.42%	28.55%	16.56%	0.16
<b>Kevic and Fritz-II</b>	<b>15.11%</b>	<b>28.28%</b>	<b>37.17%</b>	<b>21.13%</b>	<b>0.21</b>
Rocchio-I	12.54%	22.97%	31.10%	18.46%	0.17
Rocchio-II	15.92%	28.33%	33.90%	22.21%	0.21
Rahman and Roy	16.55%	31.29%	39.94%	23.51%	0.23
<b>STRICT</b>	<b>*21.95%</b>	<b>*38.07%</b>	<b>*46.43%</b>	<b>*28.99%</b>	<b>*0.29</b>

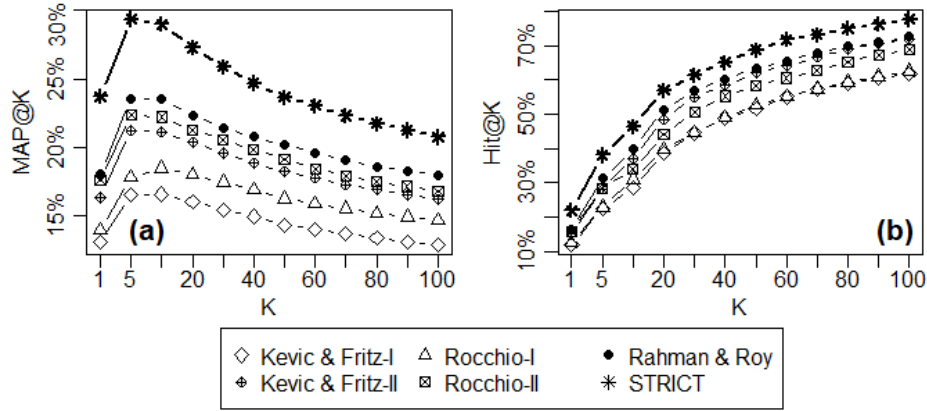
\* = Significant difference between proposed and existing techniques



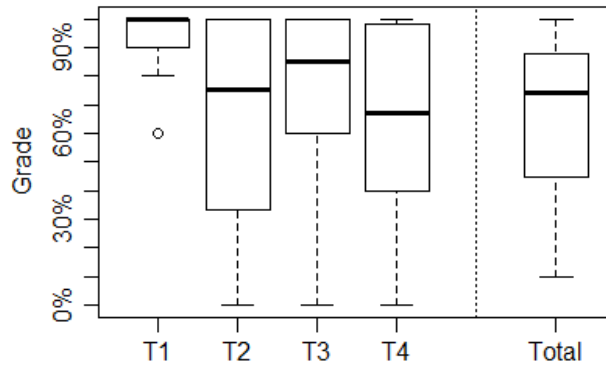
**Figure 3.9:** Comparison between queries of STRICT and the queries of existing approaches in terms of their (a) Hit@10, (b) MAP@10, and (c) MRR@10

Top-1 or Top-5 results are considered, our accuracy measures are also 32% and 22% higher respectively than the state-of-the-art measures. Fig. 3.9 further contrasts the Top-10 performance between our approach and the existing approaches using box plots. We see that our median accuracy, precision and reciprocal rank are clearly higher than that of the competing approaches (e.g., Kevic and Fritz-II). Our statistical tests also report significance with *medium to large* effect sizes (i.e., all  $p$ -values  $\leq 0.05$ ,  $0.38 \leq \Delta \leq 0.53$ ) over the state-of-the-art.

While the above analysis is based on Top-10 retrieved documents only, we further compare our approach against the existing approaches using Top-100 results. In particular, we collect accuracy and precision of each technique for top 1 to 100 documents, and compare our measures with the state-of-the-art measures. From Fig. 3.10, we see that our precision and accuracy are clearly the highest. Kevic and Fritz-II achieves a maximum precision of 21% and a maximum accuracy of 72%. On the contrary, our approach delivers 29% precision and 78% accuracy which are 38% and 8% higher respectively. Our statistical tests also report significant difference (i.e., all  $p$ -values  $\leq 0.05$ ,  $0.32 \leq \Delta \leq 0.97$ ) between our precision or accuracy and those from the state-of-the-art approaches. All these evidences clearly demonstrate the superiority of our approach over the existing approaches including the state-of-the-art.



**Figure 3.10:** Comparison between queries of STRICT and queries from the existing approaches in terms of (a) MAP@K and (b) Hit@K



**Figure 3.11:** Stage I - Distribution of the grades for study tasks

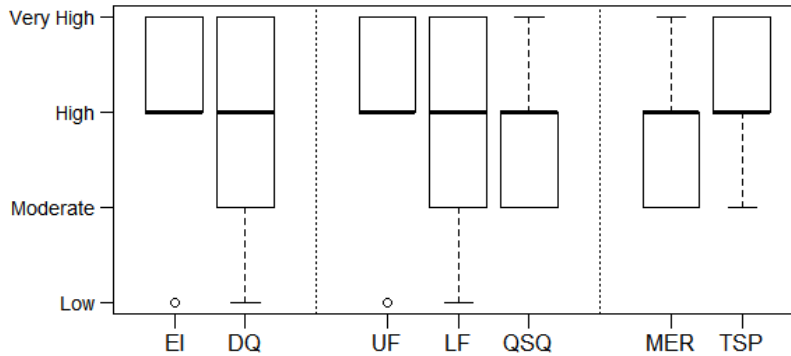
**Summary of RQ<sub>6</sub>:** Our approach clearly outperforms the existing approaches including the state-of-the-art in the retrieval of relevant source code documents (ground truth). STRICT achieves **25%** higher accuracy, **37%** higher precision and **38%** higher reciprocal rank than that of the state-of-the-art.

### 3.4.6 Evaluation of Working Prototype

Empirical evaluations using 2,885 bug reports and two evaluation dimensions clearly demonstrate the superiority of our approach over the state-of-the-art. Despite such strong evidences, we further evaluate our working prototype with a user study involving 25 participants (7 graduate students + 18 fourth year undergraduate students, from the Department of Computer Science, University of Saskatchewan).

**User Study Design:** We design a user study where participants use the proposed prototype for problem solving. Our study was conducted in two stages. In the first stage, participants were instructed to perform four different tasks<sup>3</sup> targeting concept location. The goal was to find out the best search queries possible from the change requests for concept location using partial and complete supports from the prototype. Each of the

<sup>3</sup><https://goo.gl/uBpYe8>



**Figure 3.12:** Stage II - User evaluation of the proposed prototype in terms of **EI**=Ease of Installation, **DQ**=Documentation Quality, **UF**=Usefulness of Features, **LF**=Likelihood of Features, **QSQ**=Quality of Suggested Queries, **MER**=Manual Effort Reduction, **TSP**=Time Saving Potential

participants completed the tasks using our prototype and then submitted their results. In the second stage, the participants completed a questionnaire<sup>4</sup> based on their work experience. They were asked 11 questions about the installation, usability, perceived usefulness and benefits of the proposed prototype.

**Participant Selection:** In the first stage, 18 undergraduate students from the Department of Computer Science, University of Saskatchewan took part in our study. In the second stage, six graduate and undergraduate students chose to fill in the questionnaire. Each of these students were enrolled in *CMPT 470/816: Advanced Software Engineering*, a graduate level course offered at the University of Saskatchewan. Thus, they received basic training on concept location and software change life cycle, which made them suitable candidates for our study.

**Study Tasks and Ground Truth:** We choose 10 change requests from *ecf* system for our study. Each request contains only regular texts and was already marked as RESOLVED. We instruct the participants to perform four tasks with each of these requests – **T1**: execute the available tool commands, **T2**: get the best query from STRICT, **T3**: determine the result rank improvement over the baseline, and **T4**: construct the optimal query with STRICT using manual keyword tweaking. We also perform each of these tasks ourselves, consult with the change requests and their corresponding changed source documents, and then construct the ground truth.

**User Study Results (Stage-I):** Once the first stage of our study was over, we evaluated the task results (submitted by the participants) against our ground truth. Each of these submissions was graded between 0 to 100. Fig. 3.11 shows the distribution of grades for each of the tasks scored by 18 participants. We see that almost each participant was able to successfully execute the available commands and produce the expected results (i.e., T1). This suggests that our prototype is *functional*. Participants scored a median grade of 75%–85% for T2 and T3. This suggests that they were able to spot the *differences* in query quality between baseline and STRICT, and also found the prototype’s documentation *helpful*. In the fourth task, the participants used STRICT queries as a starting point, and then attempted to identify the best possible queries

<sup>4</sup><https://goo.gl/ztoLp7>

using manual keyword tweaking. Their median grade ( $\approx 70\%$ ) suggests that our prototype significantly *helped* them in such attempts (i.e., also check TSP in Fig. 3.12). Finally, the mean grade of  $\approx 70\%$  and a median grade of 74% indicate that on average, each participant did pretty well and 50% of them did very well in performing all four given tasks. It should be noted that three students performed poorly, and scored less than 25%. Based on follow-up communications with them, we discovered that they either misunderstood the tasks or possibly were not sincere enough. Alternatively, our provided tasks could be too difficult for them technically. However, majority of the participants were successful in performing the given tasks. Such findings indicate that, on average, the participants found our prototype *working* and *useful* for their tasks.

**User Study Results (Stage-II):** Although the results above clearly demonstrate the benefits of our prototype, we additionally conduct another round of investigation by inviting the participants. We asked them 11 questions on the ease of tool installation (EI), documentation quality (DQ), overall usefulness (UF), liking for the tool features (LF), quality of our suggested queries (QSQ) and the tool’s potential for manual effort reduction (MER) or saving time (TSP) in the concept location process. Figure 3.12 summarizes our findings from the survey using box plots. The participants provided responses on a scale between 1 and 5 where 1 represents the *most negative*, 3 represents *neutral* and 5 refers to the *most positive* response about the tool. We see that the median response from the participants is *positive (high)* for each of the seven dimensions. Thus, according to the participants, installation of our prototype is very easy (i.e., EI), our provided features are likeable (i.e., UF, LF) and the suggested queries are of high quality (i.e., DQ). Besides, our tool has the potential for reducing manual efforts (i.e., MER) and spent time (i.e., TSP) during concept location. Furthermore, **66%** of the participants preferred our suggested queries over the baseline ones in their task. A few students did not put much time and efforts in the first stage, and misunderstood the task requirements, which was also reflected in their survey responses. However, majority of the participants used our tool successfully for a problem solving such as concept location. In short, all the positive responses above indicate a high potential of our approach for its possible applications in practice.

**Summary of RQ<sub>7</sub>:** Participants found our prototype **useful** in locating concepts within the source code, and most of them scored *high grades* in their tasks. Our prototype is **easy** to install, our queries are of *high quality* than baseline, and they have the potential for **reducing human efforts** and **spent time** during concept location.

### 3.5 Threats to Validity

We identify a few threats to the validity of our findings in this work. We not only discuss these threats but also outline the means that were adopted to mitigate them as follows:

**Threats to internal validity** relate to experimental errors and biases [272]. Re-implementation of the existing approaches is a possible source of such threat. Due to the lack of reliable or directly applicable prototypes, both existing techniques—Kevic and Fritz [120] and Rocchio [213]—were re-implemented. These



techniques are based on two different equations with clearly stated independent and dependent variables. We implement them carefully using authors' provided parameters, and develop multiple variants using various settings. Furthermore, we ran them in our experiments multiple times, and compared with their best performance. Thus, such threat might be mitigated.

Our suggested queries have an average length of 37 keywords which is a bit lengthy compared to the queries used for traditional web or code search (e.g., 2-3 keywords) [45, 205]. However, unlike web or code search, a change request deals with multiple domain level concepts which might not be expressed properly using only a few keywords [87]. Our queries are thus longer than *title* but shorter than the *description* of a change request. Future works might attempt to reduce the query length using more sophisticated term weighting approaches.

The POS tagging (Section 3.3.3) might contain a few false positives given that preprocessed sentences are used instead of original sentences. However, its impact might be low since stemming was not performed that affects the individual words. Furthermore, the preprocessing step mostly removes the stop words, punctuation marks and digits which convey only little semantics.

The data-resampling step of our query difficulty model (Section 3.3.8) plays an important role in delivering the best candidate queries. However, our query difficulty model might be slightly biased towards the *high*-class candidates. The challenges of data imbalance might not have been handled rigorously. Future work should employ more rigorous methods for dealing with imbalanced data. In other words, while our work in this chapter produces multiple high quality candidate queries from a change request, the future work should focus on delivering the best candidate query from them more accurately but in a non-biased fashion.

**Threats to external validity** relate to the generalizability of an approach [201, 272]. So far, we experimented with eight Java-based systems. However, given our generic approaches for term weighting and the simplicity in the corpus creation (Section 3.4.2), our approach can be easily replicated for subject systems using other programming languages. It also should be noted that our approach is not constrained by programming language specific features.

**Threats to construct validity** relate to the suitability of metrics or measures adopted for the evaluation. We use *post-retrieval* metrics such as Hit@K, MAP and MRR which are widely adopted by the relevant literature on search query suggestion [53, 120, 153] and query reformulation [98, 189]. Thus, such threats are possibly mitigated.

**Threats to conclusion validity** stem from the relationship between treatment and outcome [140]. We evaluate our term weighting approaches and query suggestion performance using six research questions, and claim superiority of STRICT over the baseline and the state-of-the-art. However, these claims are substantiated with appropriate experiments, and several statistical tests such as *Wilcoxon Signed Rank* and *Cliff's Delta* are performed. We also report details (e.g., p-values,  $\Delta$ ) of the conducted tests before making any claims. Thus, such threats might also be mitigated.

## 3.6 Related Work

### 3.6.1 Search Query Suggestion & Reformulation

A number of studies in the literature attempt to support software developers in *concept/feature/concern location* tasks using search query suggestion. They apply different lightweight heuristics [120], structural analyses [49, 163] and query reformulation strategies [65, 84, 98, 99, 188, 189, 191, 226]. They also perform different query quality analyses [95, 96, 97, 192] and data mining activities [109, 109, 121, 139, 265]. However, most of these approaches (1) expect a developer to provide the initial search query which they can improve upon, and (2) their main focus is improving a given query from the change request. Unfortunately, as existing studies [83, 120, 142] suggest, preparing an initial search query is equally challenging, and those approaches do not provide much support in this regard. In this study, we propose a novel technique—**STRICT**—that suggests a list of suitable terms as an *initial search query* from a change request. Kevic and Fritz [120] consider a list of heuristics such as *frequency*, *location*, *part of speech* and *notation* of the terms in the task description, and employ a *logistic regression model* for identifying search terms from a change request. Our work is closely related to theirs, and we compare with it directly in our experiments (Section 3.4.5).

In essence, our work is also aligned with query reformulation domain since it reformulates a baseline query (i.e., change request) by discarding the low quality search terms from the query. Rocchio [213] collects top documents from the corpus returned by a given query, identifies appropriate candidate terms from these documents using term weighting (e.g., TF-IDF [114]), and then reformulates the query. Such expansion strategy serves as a popular baseline for a number of recent studies on query reformulation [84, 98, 168, 189, 192, 251]. We thus consider Rocchio’s method as a suitable candidate for our comparison as well (Section 3.4.5). Our work in this article is also significantly different from our earlier work [191] in terms of methodology and experiments. Previously, we proposed a basic graph-based term selection approach, and conducted evaluation using a limited set and variety of change requests and research questions. This work provides an improved version of that approach using not only three graph-based algorithms (*TextRank* [153], *POSRank* [53], *Weighted K-Core* [217]) but also two novel dimensions such as *query difficulty analysis* and *machine learning*. We also conduct a more extended evaluation using  $\approx 1000$  more change requests and *three* more research questions. Furthermore, we perform more in-depth analysis on our previous research questions, and also provide a *verified working prototype* for replication and reuse. We directly compare with three state-of-the-art approaches – Kevic and Fritz, Rocchio and our earlier work (Rahman and Roy), and the detailed comparison can be found in Section 3.4.5.

Haiduc et al. and colleagues conduct several studies on how to reformulate a given search query where they apply query quality analyses [95, 97] and machine learning [98]. Although their studies are closely related to ours from a technical perspective, they are also significantly different in several aspects. First of all, they make use of source code for query preparation, whereas we use change request texts. Second, they

require an initial query from the developers which is already challenging for them to prepare [120, 142]. Since our approach suggests a search query from the change request using light-weight analysis, our work has the potential to complement the existing works on query reformulation including Haiduc et al. A few studies [109, 265] suggest semantically similar query for a given query by mining comment-code mapping from a source code repository. They could also possibly perform better if the initial query is prepared carefully which our technique does, rather than the query is chosen randomly. Bassett and Kraft [49] apply structural term weighting to feature location by emphasizing on source code tokens during query formulation. However, as our finding suggests (RQ<sub>1</sub>, Section 3.4.4), source code tokens might always not be available, and thus queries based on them could be limited in performance. Chaparro et al. [65] recently analyse bug report texts, identify expected behaviour (EB), observed behaviour (OB) and steps to reproduce (S2R), and return OB as a search query from the report. Their work is also related to ours. However, their study is empirical in nature which involves significant manual analysis, and do not provide any reusable prototype. We thus could not directly compare with their work. Several other studies apply ontology [257], query-based configurations [49, 163, 164] and phrasal concepts [105, 226] in concept location. Several other studies [58, 131, 144, 151, 274] on automated query suggestion and reformulation target general-purpose or internet-scale code search, and thus, they are not closely related to ours.

### 3.6.2 Code Search Algorithm

There also exist a number of studies [37, 142, 150, 178, 210, 225] that apply various underlying algorithms to actually locate the concepts, features, concerns or bugs in the source code. They adopt static analysis, dynamic analysis or perform both analyses on the source code to identify the items of interest such as methods to be changed. Revelle et al. [210] combine information from three different processes—textual analysis, dynamic analysis and web mining—and apply *PageRank* algorithm [57] like ours. However, they apply the algorithm in a different context – ranking methods within the project source. On the contrary, we use PageRank algorithm for the search term identification from a change request. Antoniol et al. [37] first use Vector Space Model (VSM) for traceability analysis which was later improved by Zhou et al. [276] as *rVSM* for bug localization. Since then several approaches adopt VSM-based search engine for bug localization [163, 164, 192, 258] and concept location [98, 104, 189, 201]. We also similarly use a VSM-based engine namely *Lucene*<sup>5</sup> for concept location. However, as demonstrated and claimed in earlier sections, our main contribution is the suggestion of appropriate search queries from the change requests.

Thus, from a technical perspective, we (1) adapt three graph-based term weighting algorithms (TextRank, POSRank and WK-Core) in the context of concept location which are borrowed from Information Retrieval domain, and (2) then identify a list of suitable terms from each change request. We exploit not only the co-occurrences but also the syntactic dependencies and cohesion among the words for search term identification. Furthermore, we employ query quality analysis and machine learning for identifying the best search query

---

<sup>5</sup><http://lucene.apache.org/core>

from a change request. Such idea was considered by no relevant existing studies, and the experimental findings also confirm the high potential of our idea.

### 3.7 Summary

Software maintenance is costly in terms of time, money and development efforts [88]. Developers deal with thousands of change requests during the maintenance phase. Studies suggest that choosing an appropriate search query from a change request is a major challenge for the developers [120, 142, 150]. We propose a novel technique –**STRICT**– that accepts a change request as a search query, automatically identifies the suitable keywords from the request texts, and then delivers a reformulated query for concept location. In particular, our approach constructs multiple reformulation candidates from the request texts by employing three graph-based term-weighting algorithms (TextRank, POSRank and WK-Core), and then delivers the best reformulated query using query difficulty analysis and machine learning. Experiments using 2,885 change requests from eight subject systems show that our approach improves 43%–74% of the baseline queries, and achieves 26% higher accuracy, 25% higher precision and 26% higher reciprocal rank than the baseline. Comparison with three existing studies including the state-of-the-art also shows that our approach improves 16% more baseline queries, and achieves 25% higher accuracy, 37% higher precision and 38% higher reciprocal rank than the those of state-of-the-art. Our developed tool was also successfully verified by third parties, and it received an overall positive response.

Despite these inspiring instances, we notice that our approach could be limited by the content of the change requests. That is, if a change request does not contain the right search keywords in the first place, our approach cannot deliver them. Our second study in the next chapter (ACER, Chapter 4) overcomes this challenge, and reformulates a given query with complementary keywords carefully collected from the relevant source code in order to improve the concept location task.

## CHAPTER 4

# SEARCH QUERY REFORMULATION FOR CONCEPT LOCATION USING CODERANK AND SOURCE DOCUMENT STRUCTURES

Software developers address thousands of change requests during maintenance phase, which cost a significant amount of development time and efforts [88]. Our previous study (STRICT, Chapter 3) accepts a change request as a search query, identifies suitable keywords from the request texts, and then suggests a reformulated search query for concept location. Extensive empirical evaluation, validation and user study demonstrate the high potential of the approach (Section 3.4). However, STRICT could be limited due to its sole reliance on the change requests. That is, the approach might not be able to deliver appropriate search keywords if they are missing from the change requests in the first place. We overcome such a challenge with another study in this chapter. Here, we present ACER that accepts a poor search query, identifies complementary search keywords from the relevant source code (retrieved by the query), and then delivers an improved, reformulated version of the given query for the concept location task.

The rest of the chapter is organized as follows – Section 4.1 presents an overview of our study, and Section 4.2 provides our proposed technique for query reformulation for concept location. Section 4.3 discusses our conducted experiments, findings and validations, Section 4.4 identifies the threats to the validity, Section 4.5 discusses the related work, and finally Section 4.6 concludes the chapter with future work.

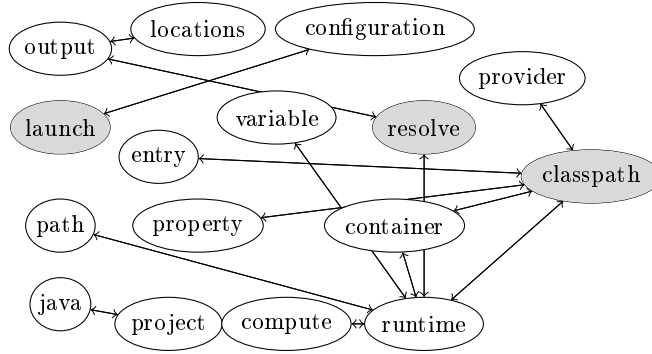
### 4.1 Introduction

Studies show that 40%–80% of the total development effort is spent in software maintenance [88, 175]. Developers deal with thousands of software issues during the maintenance [170, 220, 247]. Software issue reports (e.g., change requests, bug reports) discuss either unexpected (or erroneous) features such as software bugs or expected but non-existent features such as new software functionalities. Whether it is a bug resolution or a new feature implementation, a developer is always required to map the concept discussed in the issue report to appropriate code locations within a software project. Such a mapping task is widely known as concept location in the literature [121, 150, 188]. Developers generally choose one or more important keywords from the report texts, and then use a search method (e.g., regular expression) to locate the source code entities (e.g., classes, methods) that need to be changed. Unfortunately, as the existing studies [120, 142] report, developers regardless of their development experience perform poorly in choosing the right search queries for

concept location. According to a user study of Kevic and Fritz [120], only 12.20% of the search keywords chosen by the developers were able to locate the relevant source code during concept location. Furnas et al. [83] also suggest that there is a little chance (i.e., 10%–15%) that the developers would guess the exact words used in the source code. Thus, appropriate search query construction for concept location task is a major challenge. One way to assist the developers in this regard is to automatically reformulate their chosen queries with complementary keywords.

Existing studies employ relevance feedback from developers [84], pseudo-relevance feedback from Information Retrieval methods [98], and machine learning [98, 164] for the query reformulation tasks. They also make use of the context of query keywords from source code [104, 109, 188, 231, 265], text retrieval configurations [98, 164], and quality of queries [96, 97] in suggesting the reformulated queries. Gay et al. [84] capture explicit feedback on document relevance from the developers, and then suggest reformulated queries using Rocchio’s expansion [213]. Haiduc et al. and colleagues [95, 96, 97, 98, 99] take quality of a given query (i.e., query difficulty) into consideration, and suggest the best reformulation strategy for the query using machine learning. While all these above techniques are reported to be novel or effective, most of them also share several limitations. First, source code documents contain both structured items (e.g., method signatures, field signatures) and unstructured items (e.g., code comments). Unfortunately, many of the above reformulation approaches [84, 98, 231] treat the source code documents as simple plain text documents, and ignore most of their structural aspects except the structured tokens. Since they rely on source code for constructing their reformulated queries, such an inappropriate treatment of the code might lead them to suboptimal or poor queries. In fact, Hill et al. [104] first consider source document structures, and suggest natural language phrases from method signatures and field signatures for concern location. However, since they apply only simple textual matching between given queries and the signatures, their suggested phrases are subject to the quality of not only the given queries and but also of the identifier names from the signatures. Second, many of the above approaches often directly apply traditional metrics of term importance (e.g., TF-IDF [98, 213]) to source code, which were originally targeted for unstructured regular texts such as news articles [114]. Thus, they might fail to identify the appropriate search terms from the structured source code documents, which could badly hurt their reformulated search queries.

In this chapter, we propose and design a novel query reformulation technique – ACER – that automatically reformulates a poor search query for concept location task. We first introduce a novel graph-based term weighting algorithm – *CodeRank* – for identifying important terms from source code. CodeRank determines importance of a term not only by capturing its occurrences within the structured tokens (e.g., camel case tokens) but also by exploiting its co-occurrences with other terms across various salient entities (e.g., method signatures) within the code. Our technique – ACER – accepts a given search query as an input, employs CodeRank algorithm on the source code documents returned by the query, and develops multiple reformulation candidates using two important structural contexts from the code – method signatures and



**Figure 4.1:** An example term graph generated by CodeRank for the source code of Fig. 4.2

field signatures. Then it performs query difficulty analysis and machine learning [96, 98], and delivers the best candidate as a reformulated query for the given poor query.

Table 4.1 shows an example change request [6] submitted for `eclipse.jdt.debug` system, and it refers to “debugger source lookup” issue of Eclipse IDE. Let us assume that the developer chooses a few important keywords from the title of the change request, and formulates a generic search query—“debugger source lookup.” Unfortunately, the query does not perform well, and returns the first correct result at the 79<sup>th</sup> position of the result list. Further extension—“debugger source lookup work variables”—also does not help, and returns the result at the 77<sup>th</sup> position. The existing technique – RSV [62]— extends the query as follows—“debugger source lookup work variables *launch configuration jdt java debug*”—where the new terms are collected from the project source code using TF-IDF based term weight. This query returns the correct result at the 30<sup>th</sup> position which is also far from ideal unfortunately. The query of Sisman and Kak [231]—“debugger source lookup work variables *test exception suite core code*”—also returns the correct result at the 51<sup>st</sup> position. On the other hand, our suggested query—“debugger source lookup work variables *launch debug problem resolve required classpath*”—returns the correct result at the 2<sup>nd</sup> position which is highly promising. We first collect structured tokens (e.g., `resolveRuntimeClasspathEntry`) from method signatures and field signatures of the source code (e.g., Fig. 4.2), and split them into simpler terms (e.g., `resolve`, `Runtime`, `Classpath` and `Entry`). The underlying idea is that such signatures often encode high level intents and important domain concepts whereas the rest of the code focuses on more granular level implementation details, and thus possibly contains more noise [104, 226]. We develop individual term graph (e.g., Fig. 4.1) based on term co-occurrences from each signature type, apply CodeRank term weighting, and extract multiple candidate reformulations with the highly weighted terms (e.g., gray coloured, Fig. 4.1). Then we analyze the quality of the candidates using their quality measures [96], apply machine learning, and suggest the best reformulation to the given query. Thus, our technique (1) first captures salient terms from the source code documents by leveraging their structural aspect with an appropriate term weighting algorithm (CodeRank), (2) generates multiple reformulation candidates from the two signatures (method signatures and field signatures), and (3) then delivers the best reformulated query using query difficulty analysis and machine learning [96].

**Table 4.1:** An Example Change Request (Issue #31110, eclipse.jdt.debug)

Field	Content
Title	Debugger Source Lookup does not work with variables
Description	In the Debugger Source Lookup dialog I can also select variables for source lookup. (Advanced... > Add Variables). I selected the variable which points to the archive containing the source file for the type, but the debugger still claims that he cannot find the source.

#### An Example of Query Reformulation

Technique	Reformulated Query	QE
Baseline	{debugger source lookup work variables}	77
RSV [212]	{debugger source lookup work variables} + {launch configuration jdt java debug}	30
Sisman and Kak [231]	{debugger source lookup work variables} + {test exception suite core code}	51
<b>ACER</b> (Proposed)	{debugger source lookup work variables} + {launch debug resolve required classpath}	<b>02</b>

QE = Rank of the first correct result returned by the query

Experiments using 1,675 baseline queries from eight open source subject systems show that our technique can improve 71% (and preserve 26%) of the baseline queries which are highly promising according to the relevant literature [62, 98, 164]. Our suggested queries return correct results for 64% of the queries in the Top-100 results. Our findings report that *CodeRank* is a more effective term weighting method than the traditional alternatives (e.g., TF, TF-IDF) for search query reformulation in the context of source code. Our findings also suggest that structure of a source code document is an important paradigm both for term weighting and for search query reformulation. Comparison with five closely related existing approaches [62, 98, 104, 213, 231] not only validates our empirical findings but also demonstrates the superiority of our technique. Thus, our work makes the following contributions:

- (a) A novel term weighting method for source code –CodeRank– that identifies the important keywords from a list of given source code entities (e.g., classes, methods).
- (b) A novel query reformulation technique –ACER– that accepts a poor search query, identifies complementary keywords from the source code using CodeRank, source document structures, query quality analysis and machine learning, and then delivers an improved, reformulated query for concept location.
- (c) Comprehensive evaluation of the proposed technique using 1,675 baseline queries from eight open source subject systems.
- (d) Comparison with five closely related existing approaches from the literature.



```

public static IRuntimeClasspathEntry[] resolveRuntime
ClasspathEntry(IRuntimeClasspathEntry entry,
IJavaProject project) throws CoreException {
switch (entry.getType()) {
case IRuntimeClasspathEntry.PROJECT:
// if the project has multiple output locations,
they must be returned
IResource resource = entry.getResource();
if (resource instanceof IProject) {
IJavaProject jp = JavaCore.create((IProject)
resource);
if (jp.exists() && jp.getProject().isOpen()) {
IRuntimeClasspathEntry[] entries =
resolveOutputLocations(jp);
}
}
break;
}
}
}

```

Figure 4.2: Source code used for automated query reformulation

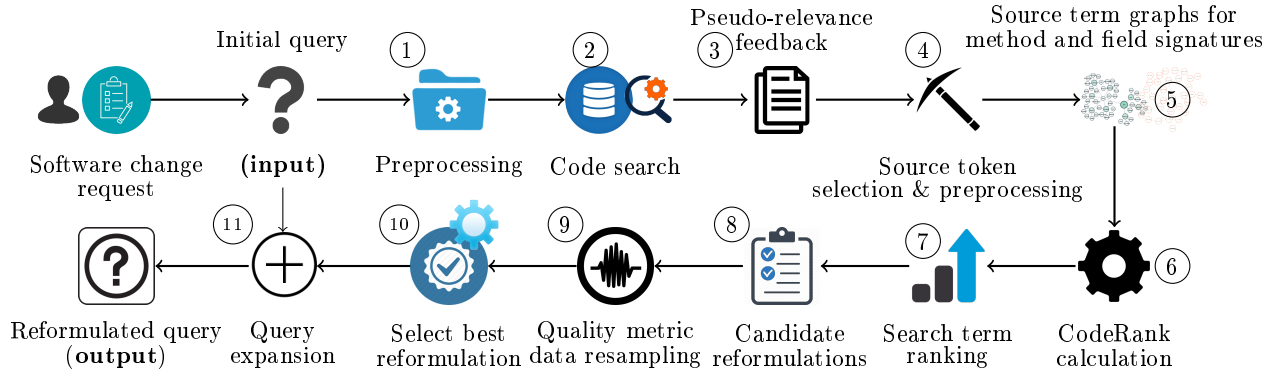


Figure 4.3: Schematic diagram of the proposed query reformulation technique—ACER

## 4.2 ACER: Automated Query Reformulation with CodeRank and Document Structures for Concept Location

Fig. 4.3 shows the schematic diagram of our proposed technique—ACER—for automatic query reformulation. We use a novel graph-based metric of term importance—*CodeRank*—for source code, and apply source document structures, query quality analysis and machine learning for query reformulation for concept location. We define *CodeRank* and discuss different steps of ACER in the following sections.

### 4.2.1 Pseudo-relevance Feedback

In order to suggest meaningful reformulations to an initial query, feedback on the query is required. Gay et al. [84] first reformulate queries based on explicit feedback from the developers. Although such feedback could be useful, gathering them is often time-consuming and sometimes infeasible. Hence, a number of recent

studies [62, 98, 188, 191] apply pseudo-relevance feedback as a feasible alternative. The top ranked results returned by the code search tool for an initial query are considered as the *pseudo-relevance feedback* for the query. We first refine an initial query by removing the punctuation marks, numbers, special symbols and stop words (Step 1, Fig. 4.3). Then we collect the Top-K (i.e.,  $K = 10$ , best performing heuristic according to our experiments) search results returned by the query, and use them as the source for our candidate terms for query reformulation (Steps 2, 3, Fig. 4.3).

## 4.2.2 Source Token Selection for Query Reformulation

**Global Query Contexts:** Pseudo-relevance feedback on an initial query provides a list of relevant source documents where one or more terms from the query generally occur. Sisman and Kak [231] choose such terms for query reformulation that frequently co-occur with the initial query terms within a fixed window size in the feedback documents. Hill et al. [104] consider presence of the query terms in method signatures or field signatures as an indicator of their relevance, and suggest natural language phrases from them as reformulated queries. Both reformulation approaches are highly subject to the quality of the initial query due to their imposed constraints—co-occurrences with query terms [231] and textual similarity with query terms [104]. Rocchio [213] determines importance (i.e., TF-IDF) of a candidate term across all the feedback documents, and suggests the top-ranked terms for query reformulation. Carmel et al. [61] suggest that a single natural language query might focus on multiple topics, and different parts of the returned results might cover different topics. That is, the same candidate term is not supposed to be important across all the feedback documents. In other words, accumulating term weight across all the documents might not always return the most appropriate terms for query reformulation. Such sort of calculation might add unnecessary noise to the term weight from the unrelated topics. Hence, we consider all the feedback documents as a *single body of structured texts* which acts as a “*global context*” for the query terms. Thus, with the help of an appropriate term weighting method, the terms representing the most dominant topic across the feedback documents (i.e., also in the initial query) could simply stand out, and could be chosen for reformulation.

**Candidate Token Mining:** Developers often express their intent behind the code and encode domain related concepts in the identifier names and comments [94]. However, code comments are often inadequate or outdated [246]. All identifier types also do not have the same level of importance. For example, while the signature of a method encodes the high level intent for the method, its body focuses on granular level implementation details and thus possibly contains more noisy terms [104]. In fact, Hill et al. [104] first analyze method signatures and field signatures to suggest natural language phrases as queries for code search. In the same vein, we thus also consider method signatures (*msig*) and field signatures (*fsig*) as the source for our candidate reformulation terms. We extract structured identifier names from these signatures using appropriate regular expressions [211] (Step 4, Fig. 4.3). Since different contexts of a source document might convey different types or levels of semantics (i.e., developers’ intent), we develop a separate candidate token

set ( $CT_{sig}$ ) for each of the two signature types ( $sig \in \{msig, fsig\}$ ) from the feedback documents ( $\forall d \in D_{RF}$ ) (i.e., relevant source documents) as follows:

$$CT_{sig} = \bigcup_{\forall d \in D_{RF}} \{\exists t \in T_{sig} \mid structured(t) \wedge T_{sig} = sig(d)\} \quad (4.1)$$

Here  $sig(d)$  extracts all tokens from method signatures or field signatures, and  $structured(t)$  determines whether the token  $t \in T_{sig}$  is structured or not. Although we deal with Java source code in this research where the developers generally use camel case tokens (e.g., `MessageType`) or occasionally might use same case tokens (e.g., `DECIMALTYPE`), our approach can also be easily replicated for snake case tokens (e.g., `reverse_traversal`).

### 4.2.3 Source Code Preprocessing

**Token Splitting:** Structured tokens often consist of multiple terms where the terms co-occur (i.e., are concatenated) due to their semantic or temporal relationships [226]. We first split each of the complex tokens based on punctuation marks (e.g., dot, braces) which returns the individual tokens (Step 4, Fig. 4.3). Then each of these tokens is splitted using a state-of-the-art token splitting tool—*Samurai* [79]—given that regular expression based splitting might not be always sufficient enough. *Samurai* mines software repositories to identify the most frequent terms, and then suggests the splits for a given token. We implement *Samurai* in our working environment where our subject systems (Section 4.3.1) are used for mining the frequent terms, and the author’s provided prefix and suffix lists [19] are applied to the splitting task.

**Stop word and Keyword Removal:** Since our structured tokens comprise of natural language terms, we discard stop words from them as a common practice (Step 4, Fig. 4.3). We use a standard list [25] hosted by Google for stop word removal. Programming keywords can often be considered as the equivalence of stop words in the source code which are also discarded from our analysis. Since we deal with Java source code, the keywords of Java are considered for this step. As suggested by earlier study [98], we also discard insignificant source terms (i.e., having word length < 3) from our analysis.

**Stemming:** It extracts the root (e.g., “send”) out of a word (e.g., “sending”). Although existing studies suggest contradictory [120, 220] or conflicting [106] evidences for stemming with the source code, we investigate the impact of stemming with RQ<sub>4</sub> where Snowball stemmer [106, 176] is used for stemming.

### 4.2.4 Source Term Graph Development

Once candidate tokens are extracted from method signatures and field signatures, and are splitted into candidate terms, we develop source term graphs (e.g., Fig. 4.1) from them (Step 5, Fig. 4.3). Developers often encode their intent behind the code and domain vocabulary into the carefully crafted identifier names where multiple terms are concatenated. For example, the method name—`getChatRoomBots`—looks like a natural

language phrase—“*get chat room bots*”—when splitted properly. Please note that each of these three terms—“*chat*”, “*room*” and “*bots*”—co-occur with each other to convey an important concept— a robotic technology, and thus, they are semantically connected. On the other hand, the remaining term—“*get*”—co-occurs with them due to a temporal relationship (i.e., develops a verbal phrase). Similar phrasal representations (refined with lexical matching) were directly returned by Hill et al. for query reformulation. However, their approach could be limited due to the added constraint (e.g., warrants query terms in signatures). We thus perform further analysis on such phrases, and exploit the co-occurrences among the terms for our graph based term weighting. In particular, we encode the term co-occurrences into connecting edges ( $E$ ) in the term graph ( $G(V, E)$ ) where the individual terms ( $V_i$ ) are denoted as vertices ( $V$ ).

$$V = \bigcup_{\forall t \in CT_{sig}} \{V_i \in splitted(t) \mid validterm(V_i)\} \quad (4.2)$$

$$E = \bigcup_{\exists V_i, V_j \in V} \{(V_i, V_j) \mid V_i, V_j \in t \wedge |i - j| = 1\} \quad (4.3)$$

Here  $splitted(t)$  returns individual terms from the token  $t \in CT_{sig}$ , and  $validterm(V_i)$  determines whether the term is valid (i.e., not an insignificant or a stop word) or not. We consider a *window size* of *two* within each phrase for capturing co-occurrences among the terms. Such window size for co-occurrence was reported to perform well by the earlier studies [53, 153, 191]. Thus, the above method name can be represented as the following edges—  $get \longleftrightarrow chat$ ,  $chat \longleftrightarrow room$ , and  $room \longleftrightarrow bots$  — in the term graph. That is, if a set of terms are frequently shared across multiple tokens from two signature types, such occurrences are represented as the high connectivity in the term graph (e.g., “*Classpath*” in Fig. 4.1).

#### 4.2.5 CodeRank Calculation

**CodeRank:** PageRank [57] is one of the most popular algorithms for web link analysis which was later adapted by Mihalcea and Tarau [153] for text documents as TextRank. In this research, we adapt our term weighting method from TextRank [53, 153, 191] for source code, and we call it *CodeRank*. To date, only traditional term weights (e.g., TF, TF-IDF [98, 213, 231]) are applied to source code which were originally proposed for regular texts [114] and are mostly based on isolated frequencies. On the contrary, CodeRank not only analyzes the connectivity (i.e., incoming links and outgoing links) of each source term, but also the relative weight of the connected terms from the graph recursively, and calculates the term weight,  $S(V_i)$ , as follows (Step 6, Fig. 4.3):

$$S(V_i) = (1 - \psi) + \psi \sum_{j \in In(V_i)} \frac{S(V_j)}{|Out(V_j)|} \quad (0 \leq \psi \leq 1) \quad (4.4)$$

Here,  $In(V_i)$ ,  $Out(V_j)$ , and  $\psi$  denote the vertices to which  $V_i$  is connected through incoming links, the vertices to which  $V_j$  is connected through outgoing links, and the damping factor respectively. As shown earlier using the example–`getChatRoomBots`, co-occurred terms complement each other with their semantics which are represented as bi-directional edges in the term graph. Thus, each ( $V_i$ ) of the vertices from the graph has equal number of incoming links and outgoing links, i.e.,  $in-degree(V_i)=out-degree(V_i)$ .

**Parameters and Configurations:** Brin and Page [57] consider damping factor,  $\psi$ , as the probability of randomly choosing a web page in the context of web surfing by a random web surfer. That is,  $1 - \psi$  is the probability of jumping off that page by the surfer. They use a well-tested value of 0.85 for  $\psi$  which was later adopted by Mihalcea and Tarau [153] for text documents. Similarly, we also use the same value of  $\psi$  for *CodeRank* calculation. Each of the vertices is assigned to a default value (i.e., base term weight) of 0.25 (as suggested by earlier studies [57, 153]) with which *CodeRank* is calculated. It should be noted that the base weight of a vertex does not determine its final weight when PageRank based algorithms are applied [153]. *CodeRank* adopts the underlying mechanism of recommendation or votes [153, 191] for term weighting. That is, each vertex feeds off from the scores of surrounding connected vertices from the graph in terms of recommendation (i.e., incoming edges). PageRank generally has two modes of computation–*iterative* version and *random walk* version. We use the iterative version for *CodeRank*, and the computation iterates until the weights of the terms converge below a certain threshold or they reach the maximum iteration limit (i.e., 100 as suggested by Blanco and Lioma [53]). As applied by earlier studies [53, 153], we also apply a heuristic threshold of 0.0001 for the convergence checking. The algorithm captures importance of a source term not only by estimating its local impact but also by considering its global influence over other terms. For example, the term, “*Classpath*”, Fig. 4.1, occurs in multiple structured tokens (Fig. 4.2), complements the semantics of five other terms, and thus is highly important within the term graph (i.e., Fig. 4.1). Once the iterative computation is over, each of the terms from the graph is found with a numeric score. We consider these scores as the relative weight or importance of the corresponding terms from the source code.

#### 4.2.6 Suggestion of the Best Query Reformulation

**Candidate Reformulation Selection:** Algorithms 4 and 5 show the pseudo-code of our query reformulation technique–ACER–for concept location. We first collect pseudo-relevance feedback for the initially provided query ( $Q$ ) where Top-K source documents are returned (Lines 3–5, Algorithm 4). Then we collect method signatures and field signatures from each of the documents ( $\forall d \in D_{RF}$ ), and extract structured tokens from them. We prepare three token sets– $CT_{msig}$ ,  $CT_{fsig}$  and  $CT_{comb}$  from these signatures (Lines 6–12, Algorithm 4, Step 4, Fig. 4.3) where  $CT_{comb}$  combines tokens from both signatures. Then we perform limited natural language preprocessing on each token set where *Samurai* algorithm [79] is used for token splitting. We develop separate term graph for each of these token sets where individual terms are represented as vertices, and term co-occurrences are encoded as connecting edges (Lines 3–7, Algorithm 5, Step 5, Fig. 4.3). We apply *CodeRank* term weighting to each of the graphs which provides a ranked list of terms based

on their relative importance. Then we select Top-K (e.g.,  $K = 10$ ) important terms from each of the three graphs, and prepare three reformulation candidates (Lines 8–12, Algorithm 5, Steps 6, 7, 8, Fig. 4.3).

---

**Algorithm 4** ACER: Proposed Query Reformulation

---

```

1: procedure ACER( $Q$ ) ▷  $Q$ : initial search query
2:    $L \leftarrow \{\}$  ▷ list of best reformulation query terms
3:   ▷ collecting pseudo-relevance feedback for  $Q$ 
4:    $Q_{pp} \leftarrow \text{preprocess}(Q)$ 
5:    $D_{RF} \leftarrow \text{getRelevanceFeedback}(Q_{pp})$ 
6:   ▷ collecting candidate source tokens from signatures
7:   for SourceDocument  $d \in D_{RF}$  do
8:      $CT_{msig} \leftarrow CT_{msig} \cup \text{getMethodSigTokens}(d)$ 
9:      $CT_{fsig} \leftarrow CT_{fsig} \cup \text{getFieldSigTokens}(d)$ 
10:  end for
11:   $CT_{comb} \leftarrow CT_{msig} \cup CT_{fsig}$ 
12:   $CT_{all} \leftarrow \{CT_{msig}, CT_{fsig}, CT_{comb}\}$ 
13:  for TokenList  $CT_{sig} \in CT_{all}$  do
14:     $QR[sig] \leftarrow \text{getQRCandidate}(CT_{sig})$ 
15:  end for
16:  ▷ suggesting the best reformulated query for  $Q$ 
17:   $QD \leftarrow \text{resample}(\text{getQueryQualityMetrics}(QR))$ 
18:   $QR_{best} \leftarrow \text{getBestCandidateUsingML}(QR, Q_{pp}, QD)$ 
19:   $L \leftarrow \text{combine}(Q_{pp}, QR_{best})$ 
20:  return  $L$ 
21: end procedure

```

---

**Selection of the Best Reformulation:** Haiduc et al. [98] argue that the same type of reformulation (i.e., addition, deletion or replacement of query terms) might not be appropriate for all given queries. In the same vein, we argue that query reformulations from different contexts of the source document (e.g., method signature, field signature) might have different level of effectiveness given that they embody different level of semantics and noise. That means, one or more of the reformulation candidates could improve the initial query, but the best one should be chosen carefully for useful recommendation.

Haiduc et al. [96] suggest that quality of a query with respect to the corpus could be determined using four of its statistical properties— *specificity*, *coherency*, *similarity* and *term relatedness*—that comprise of 21 metrics [60]. They apply machine learning on these properties, and separate high quality queries from low quality ones. We thus also similarly apply machine learning on our reformulation candidates (and their

metrics), and develop classifier model(s) where *Classification And Regression Tree* (CART) is used as the learning algorithm [96]. Since only the best of the four reformulation candidates (i.e., including baseline) is of our interest, the training data was inherently skewed. We thus perform *bootstrapping* (i.e., random resampling) [116, 237] on the data multiple times (e.g., 50) with 100% sample size and replacement (Step 9, Fig. 4.3), train multiple models using the sampled data, and then record their output predictions. Then, we average all the predictions for each test instance from all models, and determine their average probability of being the best candidate reformulation. Thus, we identify the best of the four candidates using our models, and suggest the best reformulation to the initial query (Lines 16–20, Algorithm 4, Steps 10, 11, Fig. 4.3). Bassett and Kraft [49] suggest that repetition of certain query terms might improve retrieval performance of the query. If none of the candidates is likely to improve the initial query according to the quality model (i.e., baseline itself is the best), we repeat all the terms from the initial query as the reformulation.

---

**Algorithm 5** getQRCandidate: Get a candidate reformulation

---

```

1: procedure GETQRCANDIDATE( $CT_{sig}$ )    ▷  $CT_{sig}$ : extracted candidate tokens from the signatures  $sig$ 
2:    $QR_{sig} \leftarrow \{\}$                                      ▷ candidate query reformulation
3:   ▷ extracting terms and their co-occurrences
4:    $ST_{sig} \leftarrow \text{preprocess}(\text{Samurai}(CT_{sig}))$ 
5:    $CO_{sig} \leftarrow \text{getTermCo-occurrences}(ST_{sig}, CT_{sig})$ 
6:   ▷ developing term graph from token set
7:    $G_{sig} \leftarrow \text{developTermGraph}(ST_{sig}, CO_{sig})$ 
8:   ▷ calculating CodeRank using the graph
9:    $CR_{sig} \leftarrow \text{normalize}(\text{calculateCodeRank}(G_{sig}))$ 
10:  ▷ getting candidate reformulated query
11:   $QR_{sig} \leftarrow \text{getTopKTerms}(\text{sortByValue}(CR_{sig}))$ 
12:  return  $QR_{sig}$ 
13: end procedure

```

---

**Working Example:** Let us consider the query—{debugger source lookup work variables}—from our running example in Table 4.2. Our term weighting method—*CodeRank*—extracts three candidate reformulations from method signatures and field signatures. We see that different candidates have different level of effectiveness (i.e., rank 02 to rank 16), and in this case, the candidate from the method signatures ( $QR_{msig}$ ) is the most effective. Our technique—ACER— not only prepares such candidate queries from various contexts (using a novel term weighting method) but also suggests the best candidate ( $QR_{best}$ ) for query reformulation. The reformulated query—{debugger source lookup work variables *launch debug resolve required classpath*}— returns the first correct result at the top position (i.e., rank 02) of the result list which is highly promising. Such effective reformulations are likely to reduce a developer’s effort during software change implementation.

**Table 4.2:** A Working Example (Bug #31110, eclipse.jdt.debug)

Source	Query Terms	QE
Bug Title	Debbgger Source Lookup does not work with variables	72
Initial Query ( $Q$ )	{debugger source lookup work variables}	77
$Q'_{msig}$	$Q_{pp} \cup (QR_{msig}=\{\text{launch debug resolve required classpath}\})$	<b>02</b>
$Q'_{fsig}$	$Q_{pp} \cup (QR_{fsig}=\{\text{label classpath system resolution launch}\})$	<b>06</b>
$Q'_{comb}$	$Q_{pp} \cup (QR_{comb}=\{\text{java type launch classpath label}\})$	16
$QR_{best} = \text{getBestCandidateUsingML}(QR_{msig}, QR_{fsig}, QR_{comb}, Q_{pp}, QD)$		
$Q'_{ACER}$	$Q_{pp} \cup QR_{best}$	<b>02</b>

**QE** = Query Effectiveness, rank of the first correct result returned by the query

### 4.3 Experiment

Although pre-retrieval methods (e.g., coherency, specificity [96]) are lightweight and reported to be effective for query quality analysis, post-retrieval methods are more accurate and more reliable [98]. Existing studies [98, 164, 191, 220] also adopt these methods widely for evaluation and validation. We evaluate our term weighting method and query reformulation technique using 1,675 baseline queries and three performance metrics. We also compare our technique with five closely related existing techniques [62, 98, 104, 213, 231]. We thus answer five research questions using our experiments as follows:

- **RQ<sub>1</sub>:** Does query reformulation of ACER improve the baseline queries significantly in terms of query effectiveness and retrieval performance?
- **RQ<sub>2</sub>:** Does CodeRank perform better than traditional term weighting methods (e.g., TF, TF-IDF) in identifying effective search terms from the source code?
- **RQ<sub>3</sub>:** Does employment of document structure improve ACER’s suggestion on good quality search terms from the source code?
- **RQ<sub>4</sub>:** How stemming, query length, and relevance feedback size affect the performance of our technique?
- **RQ<sub>5</sub>:** Can ACER outperform the existing query reformulation techniques from the literature in terms of effectiveness and retrieval performance of the queries?

#### 4.3.1 Experimental Dataset

**Data Collection:** We collect a total of 1,675 change requests from eight open source subject systems (i.e., five *Eclipse* systems and three *Apache* systems) for our experiments. Table 4.3 shows the experimental



**Table 4.3:** Experimental Dataset

System	#Classes	#CR	System	#Classes	#CR
eclipse.jdt.core-4.7.0	5,908	198	ecf-279.279	2,827	154
eclipse.jdt.debug-4.6.0	1,519	154	log4j-1.2.18	309	28
eclipse.jdt.ui-4.7.0	10,927	309	sling-9.0	4,328	76
eclipse.pde.ui-4.6.0	5,303	302	tomcat70-7.0.73	1,841	454

**CR**= Change requests

dataset. We first extract resolved change requests (i.e., marked as RESOLVED) from BugZilla and JIRA repositories, and then collect corresponding bug-fixing commits from GitHub version control histories of these eight systems. Such an approach was regularly adopted by the relevant literature [49, 98, 191, 231], and we also follow the same. In order to ensure a fair evaluation or validation, we discard the change requests from our dataset for which no source code files (e.g., Java classes) were changed or no relevant source files exist in the system snapshot collected for our study. We also discard such change requests that contain stack traces using appropriate regular expressions [163]. They do not represent a typical change request (i.e., mostly containing natural language texts) from the regular software users.

**Baseline Query Selection:** We select the *title* of a change request as the baseline query for our experiments, as was also selected by earlier studies [98, 120, 231]. However, we discard such baseline queries that already return their first correct results within the Top-10 positions. They possibly do not need any query reformulation [98]. Finally, we ended up with a collection of 1,675 baseline queries. We perform the same preprocessing steps as were done on the source documents (Section 4.2.3), on the queries before using them for code search in our experiments.

**Goldset Development:** Developers often mention a Bug ID in the title of a commit when they fix the corresponding reported bug [43]. We collect the *changeset* (i.e., list of changed files) from each of our selected bug-fixing commits, and develop individual solution set (i.e., *goldset*) for each of the corresponding change requests. Such solution sets are then used for the evaluation and validation of our suggested queries.

**Replication:** All experimental data, intermediate results, and relevant materials are hosted online [2] for replication or third party reuse.

### 4.3.2 Corpus Indexing & Source Code Search

Since we locate concept within project source, each of the source files is considered as an individual document of the corpus [220]. We apply the same preprocessing steps on the corpus documents as were done for query reformulation (i.e., details in Section 4.2.3). We remove punctuation marks and stop words from each document. Then, we split the structured tokens, and keep both the original and the splitted tokens in the preprocessed documents. We then apply *Apache Lucene*, a *Vector Space Model (VSM)* based popular search

engine, to index all the documents and to search for relevant documents from the corpus for any given query. Such approaches and tools were widely adopted by earlier studies [98, 120, 191, 224].

### 4.3.3 Performance Metrics

**Query Effectiveness (QE):** It approximates the effort required to find out the first correct result for a query. In other words, query effectiveness is defined as the rank of the first correct result returned by the query [98, 163]. The lower the effectiveness score, the better the query is.

**Mean Reciprocal Rank (MRR):** Reciprocal rank is defined as the multiplicative inverse of query effectiveness measure [220, 276]. Mean Reciprocal Rank averages such measures for all the queries. The higher the MRR value, the better the query is.

**Top-K Accuracy:** It refers to the percentage of queries by which at least one correct result is returned within the Top-K results [239, 250, 276]. The higher the metric value is, the better the queries are.

### 4.3.4 Evaluation of ACER and CodeRank

We evaluate our technique using 1,675 baseline queries from eight subject systems and three performance metrics discussed above. We determine effectiveness and retrieval performance of our suggested reformulated queries, and then compare them with their baseline counterparts. We also contrast our term weight with traditional term weights, and calibrate our technique using various configurations.

**Answering RQ<sub>1</sub>—Effectiveness of ACER Queries:** Table 4.4 and 4.5 show the effectiveness of ACER queries. If our query returns the first correct result closer to the top position than the baseline query, then we consider that as *query improvement*, and the vice versa as *query worsening*. If both queries return their first correct results at the same position, we consider that as *query preserving*. From Table 4.4, we see that ACER can improve or preserve 97% of the baseline queries (i.e., about 71% improvement and about 26% preserving) while worsening the quality of only about 3% of the queries. All these statistics are highly promising according to the relevant literature [98, 164, 191], i.e., maximum 52% improvement reported [98], and they demonstrate the potential of our technique. When individual systems are considered, our technique provides 63%–82% improvement across eight systems. According to the quantile analysis in Table 4.4, 25% of our queries return their first correct results within the Top-10 positions for all the systems except two (i.e., Top-12 position for `log4j` and Top-21 position for `tomcat70`). Please note that only 6% of the baseline queries return their correct results within the Top-10 positions (Table 4.6). On the contrary, 25% of our queries do so for six out of eight systems, which demonstrates the potential of our technique. While query improvement ratios are significantly higher than the worsening ratios (i.e., 28 times higher), it should be noted that our technique does not worsen any of the queries for two of the systems—`log4j` and `sling`.

Table 4.5 reports further effectiveness and the extent of actual rank improvements by our suggested queries. We see that reformulations from the method signatures improve the baseline queries significantly. For example, they improve 59% of the baseline queries while worsening 38% of them. Reformulations from

**Table 4.4: Effectiveness of ACER Query against Baseline Query**

System	#Queries	Improvement						Worsening						Preserving #Preserved		
		#Improved	Mean	Q1	Q2	Q3	Min.	Max.	#Worsened	Mean	Q1	Q2	Q3		Min.	Max.
<b>ecf</b>	154	100 (64.94%)	71	8	20	58	1	654	5 (3.25%)	125	48	88	220	43	329	49 (31.82%)
<b>jdt.core</b>	198	125 (63.13%)	89	8	20	51	1	1,485	7 (3.54%)	72	16	38	132	13	195	66 (33.33%)
<b>jdt.debug</b>	154	110 (71.43%)	72	10	23	73	1	1,234	3 (1.95%)	138	48	102	265	48	265	41 (26.62%)
<b>jdt.ui</b>	309	216 (69.90%)	169	10	27	92	1	3,162	13 (4.21%)	254	39	91	368	19	1,369	80 (25.89%)
<b>pde.ui</b>	302	191 (63.25%)	143	8	33	102	1	2,304	7 (2.32%)	507	70	477	1,060	40	1,172	104 (34.44%)
<b>log4j</b>	28	23 (82.14%)	35	12	17	58	3	136	0 (0.00%)	-	-	-	-	-	-	5 (17.86%)
<b>sling</b>	76	59 (77.63%)	165	9	18	120	2	1,940	0 (0.00%)	-	-	-	-	-	-	17 (22.37%)
<b>tomcat70</b>	454	345 (75.99%)	236	21	92	291	1	1,675	22 (4.84%)	292	97	261	429	34	938	87 (19.16%)
	Total = 1,675	Avg = 71.05%							Avg = 2.51%							<b>Avg = 26.44%</b>

**jdt.core** = eclipse.jdt.core, **jdt.debug** = eclipse.jdt.debug, **jdt.ui** = eclipse.jdt.ui, **pde.ui** = eclipse.pde.ui, **Mean** = Mean rank of first correct results returned by the queries, **Q<sub>i</sub>** = *i*<sup>th</sup> quartile of all ranks considered

**Table 4.5:** Effectiveness of ACER Variants against Baseline Queries

Query Pairs	Improved (MRD)	Worsened (MRD)	p-value	Preserved
ACER <sub>msig</sub> vs. Baseline	<b>58.93%</b> (-61)	37.99% (+131)	<b>*0.007</b>	<b>3.08%</b>
ACER <sub>fsig</sub> vs. Baseline	52.51% (-51)	44.57% (+151)	0.063	<b>2.91%</b>
ACER <sub>comb</sub> vs. Baseline	<b>58.62%</b> (-51)	38.19% (+136)	<b>*0.018</b>	<b>3.20%</b>
<b>ACER vs. Baseline</b>	<b>71.05%</b> (-81)	2.51% (+104)	<b>*&lt;0.001</b>	<b>26.44%</b>

\* = Statistically significant difference between improvement and worsening, **MRD** = Mean Rank Difference between ACER and baseline queries

the field signatures are found relatively less effective. However, ACER reduces the worsening ratio to as low as 2.51%, and increases the improvement ratio up to 71%, which are highly promising. More importantly, the mean rank differences (MRD) suggest that ACER *elevates* first correct results in the ranked list by **81** positions on average for at least 71% of the queries while *dropping* them for only 3% of the queries by 104 positions. Such rank improvements are likely to reduce human efforts significantly during concept location.

**Retrieval Performance of ACER Queries:** Table 4.6 reports the comparison of retrieval performance between our queries and baseline queries. Given that most of our selected queries are difficult (i.e., no correct results within the Top-10 positions [98]), the baseline queries retrieve at least one correct result within the Top-100 positions for 56% of the cases. However, our reformulations improve this ratio to about 64%, and the improvement is statistically significant (i.e., *paired t-test*,  $p\text{-value}=0.010<0.05$ , *Cohen’s D*=0.68 (*moderate*)). Similar scenarios are observed with mean reciprocal rank as well.

**Summary of RQ<sub>1</sub>:** The reformulation offered by our approach, ACER, improves the baseline queries significantly both in terms of query effectiveness and retrieval performance. ACER improves 71% of the baseline queries with 64% Top-100 retrieval accuracy.

**Answering RQ<sub>2</sub>–CodeRank vs. Traditional Term Weighting Methods:** Table 4.7 shows the comparative analysis between CodeRank and two traditional term weights–TF and TF-IDF– which are widely used in the text retrieval contexts [62, 120, 213]. While TF estimates the importance of a term based on its occurrences within a document, TF-IDF additionally captures the global occurrences of the term across all the documents of the corpus [114]. On the contrary, CodeRank employs a graph-based scoring mechanism that determines the importance of a term based on its co-occurrences with other important terms within a certain context. From Table 4.7, we see that CodeRank performs significantly better than both TF (i.e., *paired t-test*,  $p\text{-value}=0.005<0.05$ ) and TF-IDF (i.e.,  $p\text{-value}<0.001$ ) in identifying important search terms from source code, especially from the method signatures. Considering the whole source code rather than signatures improves the performance of both TF (i.e., 56% query improvement) and TF-IDF (i.e., 52% query improvement). However, our term weight–CodeRank–is still better alone (i.e., 59%), and improves significantly higher (i.e.,  $p\text{-value}=1.717e-06$ ) fraction (i.e., 71%) of the baseline queries when employed with our proposed reformulation algorithm–ACER.

**Table 4.6:** Comparison of ACER’s Retrieval Performance with Baseline Queries

Query	Metric	Top-10	Top-20	Top-50	Top-100
Baseline	Top-K Accuracy	5.78%	18.91%	41.09%	56.30%
	MRR@K	0.01	0.02	0.03	0.03
ACER <sub>msig</sub>	Top-K Accuracy	10.45%	21.48%	38.12%	51.31%
	MRR@K	0.02	0.03	0.04	0.04
ACER <sub>fsig</sub>	Top-K Accuracy	7.77%	17.40%	36.25%	47.23%
	MRR@K	0.02	0.03	0.03	0.03
ACER <sub>comb</sub>	Top-K Accuracy	8.68%	20.78%	36.87%	51.75%
	MRR@K	0.02	0.03	0.03	0.04
ACER	Top-K Accuracy	<b>*14.72%</b>	<b>*31.22%</b>	<b>*49.89%</b>	<b>*63.89%</b>
	MRR@K	0.04	0.05	<b>0.06</b>	<b>0.06</b>

\* = Statistically significant difference between ACER and baseline

Fig. 4.4 shows how *CodeRank* and traditional term weights perform in reformulating the baseline queries with their (a) Top-10 and (b) Top-30 terms. We see that TF reaches its peak performance pretty quickly (i.e.,  $K = 3$ ), and then shows a stationary or irregular behaviour. That means, TF identifies frequent terms for query reformulation, and few of them (e.g., Top-3) could be highly effective. On the contrary, our method—*CodeRank*—demonstrates a gradual improvement in the performance up to Top-12 terms (i.e.,  $K=12$ , Fig. 4.4-(b)), and crosses the performance peak of TF with a large margin (i.e., *paired t-test*,  $p\text{-value}=0.004 < 0.05$ , *Cohen’s D*= $3.77 > 1.00$  (*large*)), for  $K=10$  to  $K=15$ ). *CodeRank* emphasizes on the votes from other important terms (i.e., by leveraging co-occurrences) for determining weight of a term, and as demonstrated in Fig. 4.4, this weight is found to be more reliable than TF. TF-IDF is found relatively less effective according to our investigation.

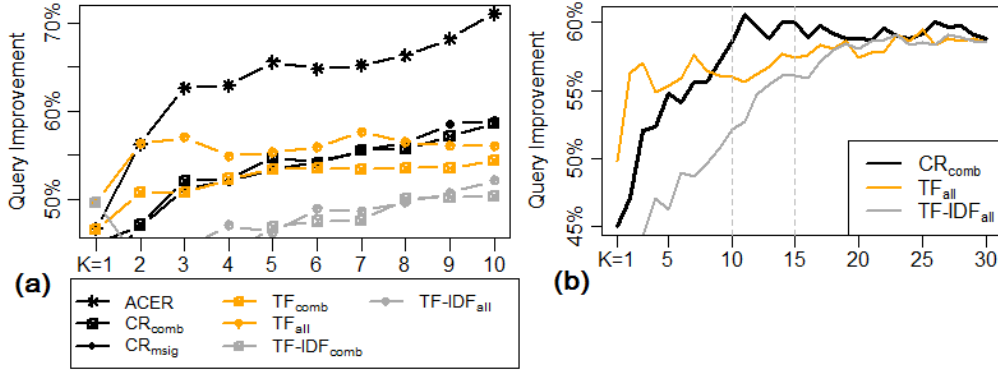
**Summary of RQ<sub>2</sub>:** *CodeRank* performs significantly better than traditional methods in identifying effective terms for query reformulation from the source code.

**Answering RQ<sub>3</sub>—Do Document Structures Matter?** While most of the earlier reformulation techniques miss or ignore the structural aspect of a source document, we consider such aspect as an important paradigm of our technique. We consider a source document as a collection of structured entities (e.g., signatures, methods, fields) [184] rather than a regular text document. Thus, we make use of method signatures and field signatures rather than the whole source code for query reformulation given that they are likely to contain more salient terms and less noise [104]. Fig. 4.5 demonstrates how incorporation of document structures into a technique could be useful for query reformulations. We see that reformulations using method signatures and field signatures improve two different sets of baseline queries, and this happens with both term weighting methods—(a) *CodeRank* and (b) TF. While these sets share about half of the queries (49%–57%), reformulations based on each signature type also improve a significant amount (i.e., 19% ( $73+136+24$ ) – 25%

**Table 4.7:** Comparison between CodeRank and Traditional Term Weights

Query Pairs	Improved	Worsened	Preserved
ACER <sub>msig</sub> vs. TF <sub>msig</sub>	<b>*58.93%</b> / 53.40%	<b>*37.99%</b> / 44.60%	3.08% / 2.00%
ACER <sub>fsig</sub> vs. TF <sub>fsig</sub>	52.51% / 51.57%	44.57% / 46.85%	2.91% / 1.57%
ACER <sub>comb</sub> vs. TF <sub>comb</sub>	<b>*58.62%</b> / 54.34%	<b>*38.19%</b> / 44.11%	3.20% / 1.54%
ACER vs. TF <sub>all</sub>	<b>*71.05%</b> / 56.01%	<b>*2.51%</b> / 41.44%	<b>*26.44%</b> / 2.55%
ACER <sub>msig</sub> vs. TF-IDF <sub>msig</sub>	<b>*58.93%</b> / 45.55%	<b>*37.99%</b> / 49.88%	3.08% / 4.57%
ACER <sub>fsig</sub> vs. TF-IDF <sub>fsig</sub>	52.51% / 51.06%	44.57% / 46.77%	2.91% / 2.17%
ACER <sub>comb</sub> vs. TF-IDF <sub>comb</sub>	<b>*58.62%</b> / 50.35%	<b>*38.19%</b> / 47.25%	3.20% / 2.40%
ACER vs. TF-IDF <sub>all</sub>	<b>*71.05%</b> / 52.17%	<b>*2.51%</b> / 45.13%	<b>*26.44%</b> / 2.70%

\* = Statistically significant difference between ACER measures and their counterparts

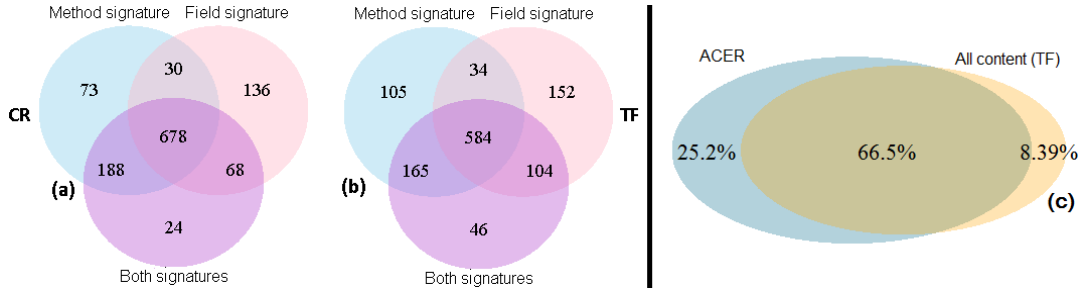


**Figure 4.4:** Comparison of query improvement between CodeRank and traditional term weights for (a) Top K=1 to 10 and (b) Top K=1 to 30 reformulated query terms

(105+152+46)) of unique baseline queries. In Fig. 4.5-(c), when these signatures (i.e., along with ACER) are contrasted with the whole source code (i.e., along with TF), we even found that the signature-based reformulations outperform the whole code-based reformulations by a large margin (i.e., (25.2%–8.39%)  $\approx$  17% more query improvement). That is, the use of the whole source code introduces additional noise, and diminishes the strength or salience of the individual structures (i.e., signatures). Most of the existing methods [84, 98, 188] suffer from this limitation. On the contrary, our technique ACER exploits document structures (i.e., signatures), and carefully chooses the best among all the candidate reformulations derived from such structures using query quality analysis and machine learning.

**Summary of RQ<sub>3</sub>:** Document structures (e.g., method signatures, field signatures) improve the suggestion of query reformulation terms from the source code.

**Answering RQ<sub>4</sub>– Impact of Stemming, Query Length, and Relevance Feedback:** From Table 4.8, we see that stemming generally degrades the effectiveness of our reformulated queries. Similar findings were also reported by earlier studies [120, 220]. Fig. 4.6 shows how (a) Top-10 and (b) Top-30 reformulation



**Figure 4.5:** Improved queries by reformulation from method signatures and field signatures using (a) CodeRank (CR) and (b) Term Frequency (TF). (c) ACER vs. TF (all content)

**Table 4.8:** Impact of Stemming on Query Effectiveness

Source	Query	Improved (MRD)	Worsened (MRD)	Preserved
Method signature	$ACER_{msig,stem}$	<b>52.66%</b> (-58)	44.73% (+127)	2.61%
	$ACER_{msig}$	<b>*58.93%</b> (-61)	<b>*37.99%</b> (+131)	3.08%
Field signature	$ACER_{fsig,stem}$	48.14% (-53)	47.47% (+151)	4.39%
	$ACER_{fsig}$	52.51% (-51)	44.57% (+151)	2.91%
Both signatures	$ACER_{comb,stem}$	52.68% (-57)	44.38% (+128)	2.94%
	$ACER_{comb}$	<b>*58.62%</b> (-51)	<b>*38.19%</b> (+136)	3.20%
Both signatures	$ACER_{stem}$	<b>68.11%</b> (-78)	5.37% (+67)	<b>26.51%</b>
	ACER	<b>71.05%</b> (-81)	<b>*2.51%</b> (+104)	<b>26.44%</b>

\* = Statistically significant difference between two measures from the same signature,

MRD = Mean Rank Difference between ACER and baseline queries

terms improve the baseline queries. We see that our reformulations perform the best (i.e., about 60% query improvement) with Top-10 to 15 search terms collected from each signature type. However, when query quality analysis [96] is employed, our technique—ACER—can improve 71% of the baseline queries with only Top-10 reformulation terms. We also repeat the same investigation with Top-30 terms, and achieved the same top performance (i.e., Fig. 4.6-(b)). Thus, our choice of returning Top-10 reformulation terms is justified. We also investigate how the size of pseudo-relevance feedback influences our performance, and experimented with Top-30 documents. We found that reformulations for ACER reach the performance peak when Top-10 to 15 feedback source documents (i.e., returned by the baseline queries) are analyzed for candidate terms. This possibly justifies our choice of Top-10 documents as the pseudo-relevance feedback.

**Summary of RQ<sub>4</sub>:** Token stemming degrades the query effectiveness of ACER. Reformulation size and relevance feedback size gradually improve the performance of ACER’s queries as long as they are below a certain threshold (i.e.,  $K = 15$ ).

**Table 4.9:** Comparison of Query Effectiveness with Existing Techniques

Technique	#Queries	Improvement						Worsening						Preserving	
		#Improved	Mean	Q1	Q2	Q3	Min.	Max.	#Worsened	Mean	Q1	Q2	Q3	Min.	Max.
Hill et al. [104]	1,675	631 (37.67%)	157	18	48	161	1	2,264	261	54	119	300	4	4,819	284 (16.96%)
Rocchio [213]	1,675	895 (53.43%)	219	15	49	188	1	4,609	333	65	170	429	3	3,489	41 (2.45%)
RSV [62]	1,675	<b>914 (54.57%)</b>	216	15	52	195	1	4,611	307	63	160	415	7	3,387	38 (2.27%)
Sisman and Kak [231]	1,675	759 (45.31%)	207	17	61	213	1	3,707	273	59	147	345	8	2,545	274 (16.36%)
Refoqus [98]	1,675	<b>895 (53.43%)</b>	217	15	51	188	1	4,609	332	65	170	429	3	3,489	43 (2.57%)
<b>Refoqus<sub>sampled</sub> [98]</b>	1,675	<b>1,154 (68.90%)</b>	156	11	33	141	1	4,609	<b>487 (29.07%)</b>	63	166	406	6	3,489	<b>34 (2.03%)</b>
ACER <sub>msig</sub>	1,675	969 (57.85%)	208	14	49	192	1	3,649	272	52	139	341	2	4,825	44 (2.63%)
ACER <sub>comb</sub>	1,675	958 (57.19%)	216	15	49	194	1	4,117	275	52	139	336	4	3,360	43 (2.57%)
<b>ACER</b>	1,675	<b>*1,169 (69.79%)</b>	156	<b>11</b>	35	130	1	3,162	260	53	140	375	13	1,369	<b>*449 (26.81%)</b>
<b>Baseline</b>	1,675	-	227	32	88	258	3	4,787	113	24	49	162	1	718	-
<b>ACER<sub>ext</sub></b>	1,755	<b>*1,192 (67.92%)</b>	149	<b>10</b>	34	124	1	3,162	<b>*48 (2.74%)</b>	50	145	327	13	1,782	<b>*515 (29.34%)</b>

Mean = Mean rank of first correct results returned by the queries,  $Q_i = i^{th}$  quartile of all ranks considered, \* = Statistically significant difference between ACER measures and their counterparts



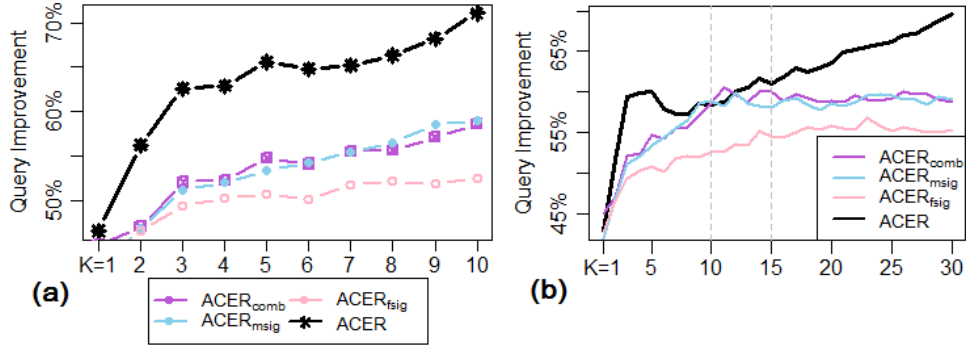


Figure 4.6: Effectiveness of ACER queries for (a) Top-10 and (b) Top-30 reformulated terms

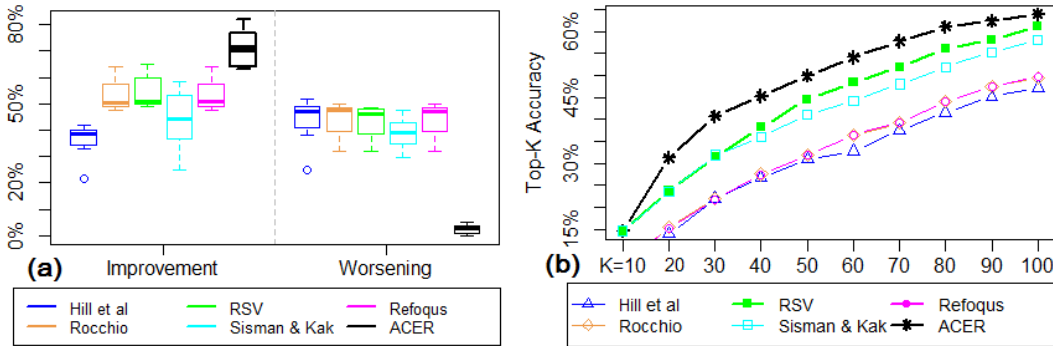


Figure 4.7: Comparison of (a) query effectiveness, and (b) retrieval performance

### 4.3.5 Comparison with Existing Approaches

**Answering RQ<sub>5</sub>:** While the empirical evaluation in terms of performance metrics above clearly demonstrates the promising aspects of our query reformulation technique, we still compare with five closely-related existing approaches [62, 98, 104, 213, 231]. Hill et al. [104] suggest relevant phrases from method signatures and field signatures as query reformulations. While Sisman and Kak [231] focus on term co-occurrences with query keywords, Rocchio [213] and RSV [62] apply TF-IDF based term weights for choosing query reformulation terms. Refoqus [98] is closely related to ours and is reported to perform better than RSV and other earlier approaches, which probably makes it the state-of-the-art for our research problem. We replicate each of Hill et al., Rocchio, RSV, Sisman and Kak, and Refoqus in our working environment by carefully following their algorithms, equations and methodologies given that their implementations are not publicly available. In the case of Refoqus, we implement 27 metrics (20 pre-retrieval [96] and 7 post-retrieval [98]) that estimate query difficulty. We develop a machine learning model using CART algorithm (i.e., as used by them) and 10-fold cross validation. Then, we use the model to return the best reformulation out of four candidates of Refoqus—*query reduction*, *Dice expansion*, *Rocchio’s expansion* and *RSV expansion*—for each baseline query. Table 4.9 and Fig. 4.7 summarize our comparative analyses.

From Table 4.9, we see that RSV and Refoqus perform better than the other existing approaches. They improve about 55% and about 53% of the baseline queries respectively. Such ratios are also pretty close to

the originally reported performances by Haiduc et al. on a different dataset, which possibly validates the correctness of our implementation. While 55% query improvement is the maximum performance provided by any of the existing approaches, our technique—ACER—improves about 70% of the baseline queries (i.e., 1% difference between Table 4.5 and Table 4.9 due to rounding error) which is significantly higher, i.e., *paired t-test*,  $p\text{-value}=6.663e-06<0.05$ , *Cohen’s D*=2.43>1.00 (*large*). Refoqus adopts a similar methodology like ours. Unfortunately, the approach is limited due to possibly the low performance of its candidate reformulations. One might argue about the data resampling step (i.e., Step 9, Fig. 4.3) of ACER for the high performance. However, we also apply data resampling to Refoqus using the same settings as ours for further investigation. We see that Refoqus<sub>sampled</sub> has a similar improvement ratio like ours, but it still worsens a significant amount of queries, 29%, compared to our query worsening ratio of 3.40%. Thus, our technique still performs better than Refoqus in the equal settings. Our quantile measures and mean ranks are more promising than those from the baseline or competing methods as reported in Table 4.9. Table 4.5 and **RQ<sub>1</sub>** also suggest that our queries have high potential for reducing human efforts. We also experiment with an extended dataset (i.e., 1,755=1,675 + 8x10) containing 80 very good queries. As reported in Table 4.9, ACER<sub>ext</sub> mostly preserves the good quality queries rather than worsening, which also demonstrates its high potential.

Fig. 4.7-(a) shows the box plots of query improvement and query worsening ratios by all the techniques under study. We see that ACER outperforms the existing techniques including the state-of-the-art [98] by a large margin. Our median improvement ratio is about 75%, which is higher than even the maximum improvement ratios of the counterparts, which demonstrates the promising aspect of ACER. Fig. 4.7-(b) shows the Top-K accuracy of the query reformulation techniques. We see that our accuracy is relatively higher than that of each of the existing approaches across various Top-K (i.e., 10–100) values. The best performing existing method is RSV. However, our performance is significantly higher than that of RSV for various K values according to statistical significance tests (i.e., *paired t-test*,  $p\text{-value}<0.05$ , *Cohen’s D*=0.34).

**Summary of RQ<sub>5</sub>:** Our technique, ACER, outperforms the state-of-the-art techniques in terms of reformulation query effectiveness, and performs significantly better than each of the existing techniques in terms of document retrieval accuracy.

## 4.4 Threats to Validity

Threats to *internal validity* relate to experimental errors and biases [272]. Although CodeRank and document structures play a major role, the data resampling step (Section 4.2.6, Step 9, Fig. 4.3) has a significant role behind the high performance of our technique. Unfortunately, to the best of our knowledge, Refoqus [98] does not have such a step. Thus, the performance comparison might look like a bit unfair. Besides, models based on data resampling are sometimes criticized for their intrinsic biases [22]. However, we apply the same data resampling step to Refoqus as well (i.e., Refoqus<sub>sampled</sub>), and demonstrate that our technique still performs

better in terms of query worsening ratio. Despite all these inspiring instances, our query difficulty models might still be slightly biased due to data imbalance problem. Future work should employ more rigorous methods for dealing with the imbalanced data.

Threats to *external validity* relate to the generalization of the obtained results [98]. All of our subject systems are Java-based. So, there might be different results with systems from other programming languages. However, we experimented with eight different systems with promising performance, and the comparison with the state-of-the-art techniques demonstrates the superiority of our approach.

## 4.5 Related Work

There exist a number of studies in the literature that reformulate a given query for concept location in the context of software change tasks. Existing studies apply relevance feedback from developers [84], pseudo-relevance feedback from IR tools [98], partial phrasal matching [104, 215], and machine learning [98, 164] to query reformulation. They also make use of context of query terms from source code [109, 188, 231, 265], text retrieval configuration [98, 164], and quality of queries [96, 97] in suggesting the reformulated queries. Hill et al. [104] consider the presence of query terms in the method or field signatures as an indicator of their relevance, and suggest natural language phrases from them as reformulated queries. Sisman and Kak [231] choose such terms for query reformulation that frequently co-occur with query terms within a fixed size of window in the code. Rocchio [213] and RSV [62] determine importance of a term using TF-IDF based metrics. Haiduc et al. [98] identify the best of four reformulation candidates for any given query using a machine learning model with 28 metrics. All these five studies are highly relevant to ours, and we directly compare with them using experiments. Readers are referred to Section 4.3.5 for comparison details.

Other related studies [187, 191, 267] explore graph-based methods for term weighting. Rahman and Roy [187, 191] simply use TextRank on *change request texts* for suggesting initial queries for concept location. Yao et al. [267] build a term augmented tuple graph and use a random walk approach to reformulate queries for structured bibliographic DBLP Data (i.e., non-source code). Ours is significantly different from these studies in the sense that we reformulate the initial queries not only by employing our term weighting method—CodeRank for *source code*, but also by applying source code document structures, query quality analysis and machine learning. Besides, their reported best performance (i.e., 58%–62% query improvement over baseline [191]) is quite lower than our performance (i.e., 71%, even with difficult queries). Given that reformulation is often performed on the initial queries, our technique can potentially complement theirs. Howard et al. [109] map method signatures to associated comments for query reformulation, and thus, might not work well with source code without comments. Our earlier work [188] exploits crowd sourced knowledge for query reformulation, and that method is also subject to the availability of a third party information source. Thus, while earlier studies adopt various methodologies or information sources, our technique not only employs

a novel and promising term weight –*CodeRank*, but also exploits structures of the source documents for identifying the best reformulation to a given query for improved concept location.

## 4.6 Summary

Software developers deal with thousands of change requests during maintenance phase. Locating a concept within the source code using the request texts is a major challenge. About 88% of the time, software developers fail to choose the right search queries from the change requests [120]. Their queries thus need to be carefully reformulated before using them for concept location. In this chapter, we propose a novel technique –ACER– that reformulates the search queries from the developers and supports the concept location task. In particular, ACER accepts a given query as input, and constructs multiple reformulation candidates from the relevant source code documents using a novel term weighting method namely *CodeRank*. Then it suggests the best reformulated query using query difficulty analysis and machine learning. Experiments with 1,675 search queries from eight systems report that our technique can improve 71% of the given queries and preserve 26% of them, which are highly promising. Comparison with five closely related existing approaches including the state-of-the-art approach not only validates our empirical findings but also demonstrates the high potential of our technique.

Our study in this chapter (ACER) and our previous study (STRICT, Chapter 3) extract important keywords from source code documents and change requests respectively, and help the developers locate the concepts of interest (e.g., program entities) within a software system. Although they are found promising for concept location, they might not be directly applicable to bug localization. Bug reports often contain highly structured entities (e.g., stack traces, test cases) as opposed to the unstructured texts in the change requests. Thus, STRICT or ACER might not be suitable for extracting appropriate keywords from these structured entities. In the next chapter, our third study (BLIZZARD, Chapter 5) overcomes this challenge. BLIZZARD accepts a bug report as a search query, employs context-aware query reformulations, and then delivers an improved, reformulated search query for bug localization even from the noisy and poor quality bug reports.

# CHAPTER 5

## SEARCH QUERY REFORMULATION FOR BUG LOCALIZATION USING REPORT QUALITY DYNAMICS & GRAPH-BASED TERM WEIGHTING

Software bugs and failures cost trillions of dollars every year [1]. One crucial step towards resolving the bugs is finding the locations of the bugs within a software system [220, 248, 276]. Our previous studies (STRICT, Chapters 3, ACER, 4) accept a change request as a search query, and then deliver a reformulated query for concept location. Although they are found promising for concept location, they might not be directly applicable to bug localization task. Bug reports often contain highly structured entities (e.g., stack traces) as opposed to the regular texts in the change requests. Thus, our previous studies that are designed for change requests might deliver sub-optimal queries from the bug reports, which hurt the bug localization performance. In this chapter, we present another study (BLIZZARD) that overcomes this challenge. BLIZZARD accepts a bug report as a search query, employs appropriate methodologies or algorithms based on the quality of the report (e.g., noisy, poor), and then delivers an improved, reformulated search query for the bug localization.

The rest of the chapter is organized as follows—Section 5.1 presents an overview of our study, and Section 5.2 describes our proposed approach for search query reformulation and bug localization. Section 5.3 discusses our evaluation, validation and answers four research questions. Section 5.4 identifies the threats to validity, Section 5.5 discusses the related work, and finally Section 5.6 concludes the chapter with future work.

### 5.1 Introduction

Despite numerous attempts for automation [41, 68, 91, 165, 278], software debugging is still largely a manual process which costs a significant amount of development time and efforts [39, 170, 262]. One of the three steps of debugging is the identification of the location of a bug in the source code, i.e., bug localization [170, 248]. Recent bug localization techniques can be classified into two broad families—*spectra based* and *Information Retrieval (IR) based* [130]. While spectra-based techniques rely on execution traces of a software system, IR-based techniques analyse shared vocabulary between a bug report (i.e., query) and the project source for bug localization [163, 276]. Performances of IR-based techniques are reported to be as good as that of spectra-based techniques, and such performances are achieved using a low cost text analysis [207, 248].

Unfortunately, recent qualitative and empirical studies [193, 248] have reported two major limitations. First, IR-based techniques cannot perform well without the presence of rich structured information (e.g., program entity names pointing to defects) in the bug reports. Second, they also might not perform well with a bug report that contains excessive structured information (e.g., stack traces, Table 5.1) [248]. One possible explanation of these limitations could be that most of the contemporary IR-based techniques [130, 167, 207, 220, 230, 249, 276] use almost verbatim texts from a bug report as a *query* for bug localization. That is, they do not perform any meaningful modification to the query except a limited natural language pre-processing (e.g., stop word removal, token splitting, stemming). As a result, their query could be either *noisy* due to excessive structured information (e.g., stack traces) or *poor* due to the lack of relevant structured information (e.g., Table 5.2). One way to overcome the above challenges is to (a) refine the noisy query (e.g., Table 5.1) using appropriate filters and (b) complement the poor query (e.g., Table 5.2) with relevant search terms. Existing studies [128, 249, 250, 268] that attempt to complement basic IR-based localization with costly data mining or machine learning alternatives can also equally benefit from such query reformulations.

In this chapter, we propose and design a novel technique –BLIZZARD– that locates software bugs from source code by employing *context-aware query reformulation* and information retrieval. Our technique (1) first determines the quality (i.e., prevalence of structured entities or lack thereof) of a bug report (i.e., query) and classifies it as either *noisy*, *rich* or *poor*, (2) then applies appropriate reformulation to the query, and (3) finally uses the improved query for the bug localization with information retrieval. Unlike earlier approaches [220, 221, 249, 276], it either refines a noisy query or complements a poor query for effective information retrieval. Thus, BLIZZARD has a high potential for improving IR-based bug localization.

To illustrate the capability of our technique in improving bug localization, we provide two examples in which it outperforms the baseline. The baseline technique that uses all terms except punctuation marks, stop words and digits from a bug report, returns its first correct result for the noisy query containing stack traces in Table 5.1 at the 53<sup>rd</sup> position. On the contrary, our technique refines the same noisy query, and returns the first correct result at the first position of the ranked list which is a significant improvement over the baseline. Similarly, when we use a poor query containing no structured entities such as in Table 5.2, the baseline technique returns the correct result at the 30<sup>th</sup> position. On the other hand, our technique improves the same poor query, and returns the result again at the first position. BugLocator [276], one of the well cited IR-based techniques, returns such results at the 19<sup>th</sup> and 26<sup>th</sup> positions respectively for the noisy and poor queries which are far from ideal.

We evaluate our technique in several different dimensions using four widely used performance metrics and 5,139 bug reports (i.e., queries) from six Java-based subject systems. First, we evaluate in terms of the performance metrics, contrast with the baseline, and BLIZZARD localizes bugs with 7%–56% higher accuracy (i.e., Hit@10), 6%–62% higher precision (i.e., MAP@10) and 6%–62% higher result ranks (i.e., MRR@10) than the baseline (Section 5.3.3). Second, we compare our technique with three bug localization techniques [220, 250, 276], and our technique can improve 19% in MAP@10 and 20% in MRR@10 over the

**Table 5.1:** A Noisy Bug Report (Issue #31637, eclipse.jdt.debug)

Field	Content
Title	should be able to cast "null"
Description	<p>When trying to debug an application the variables tab is empty. Also when I try to inspect or display a variable, I get following error logged in the eclipse log file:</p> <pre> java.lang.NullPointerException at org.eclipse.jdt.internal.debug.core.model.JDIValue.toString(JDIValue.java:362) at org.eclipse.jdt.internal.debug.eval.ast.instructions.Cast.execute(Cast.java:88) at org.eclipse.jdt.internal.debug.eval.ast.engine. Interpreter.execute(Interpreter.java:44) at org.eclipse.jdt.internal.debug.eval.ast.engine. EvaluationThread\$1\$EvaluationRunnable at org.eclipse.jdt.internal.debug.core.model.JDIThread.runEvaluation (JDIThread.java:600) .....(more)..... </pre>

#### An Example of Noise Filtration

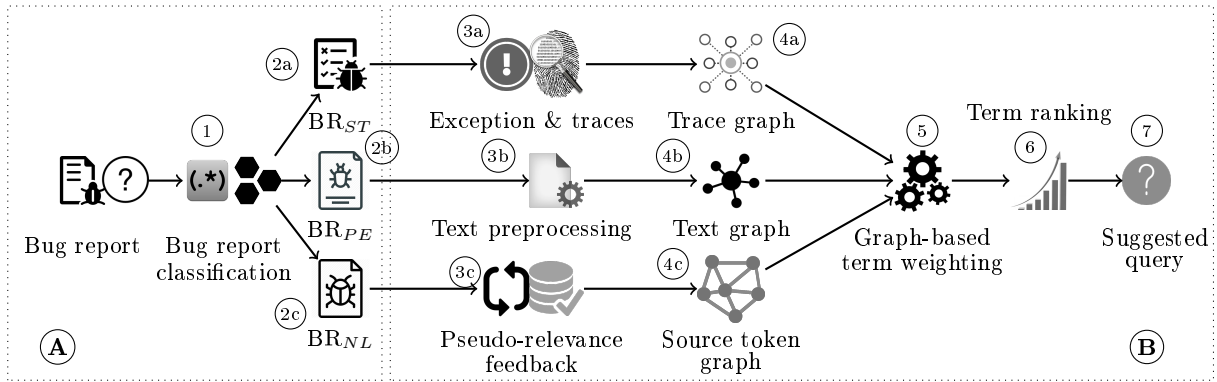
Technique	Suggested Query	QE
Baseline	{Title + Description }	53
<b>BLIZZARD</b>	NullPointerException + "Bug should be able to cast null" + {JDIValue toString execute EvaluationThread run}	<b>01</b>

state-of-the-art [250] (Section 5.3.4). Third, we also compare our approach with four state-of-the-art query reformulations techniques, and BLIZZARD improves the result ranks of 59% of the noisy queries and 39% of the poor queries which are 22% and 28% higher respectively than that of the state-of-the-art [191] (Section 5.3.4). By incorporating *report quality aspect* and *query reformulation* into IR-based bug localization, we resolve an important issue which was either not addressed properly or otherwise overlooked by earlier studies, which makes our work *novel*. Thus, our work makes the following contributions:

- (a) A novel query reformulation technique –BLIZZARD– that filters noise from and adds complementary information to the bug report, and suggests improved, reformulated search queries for bug localization.
- (b) A novel bug localization technique that locates bugs from the project source by employing quality paradigm of bug reports, query reformulation, and information retrieval.
- (c) Comprehensive evaluation of the technique using 5,139 bug reports from six open source systems and validation against seven techniques including the state-of-the-art.
- (d) A working prototype [27] with detailed experimental data for replication and third party reuses.

**Table 5.2:** A Poor Bug Report (Issue #187316, eclipse.jdt.ui)

Field	Content	
Title	[preferences] Mark Occurences Pref Page	
Description	There should be a link to the pref page on which you can change the color. Namely: General/Editors/Text Editors/Annotations. It's a pain in the a** to find the pref if you do not know Eclipse's preference structure well.	
An Example of Query Expansion		
Technique	Expanded Query	QE
Baseline	{Title + Description}	30
<b>BLIZZARD</b>	{Title + Description} + {compliance create preference add configuration field dialog annotation}	<b>01</b>



**Figure 5.1:** Schematic diagram of the proposed query reformulation technique –BLIZZARD–(A) Bug report classification and (B) Search query suggestion

## 5.2 BLIZZARD: Automated Query Suggestion using Report Quality Dynamics and Term Weighting for Bug Localization

Fig. 5.1 shows the schematic diagram of our proposed technique for automated query suggestion–BLIZZARD. Furthermore, Algorithm 6 shows the pseudo-code for BLIZZARD. We construct appropriate search queries from the bug reports by making use of the report quality dimension and graph-based term weighting, and then employ them for localizing the bugs in source code with information retrieval as follows.

### 5.2.1 Bug Report Classification

Since our primary objective with this work is to overcome the challenges posed by the different kinds of information bug reports may contain, we categorize the reports prior to bug localization. In addition to having natural language texts, a bug report typically may contain different structured elements: (1) stack traces (reported active stack frames during the occurrence of a bug, e.g., Table 5.1), and (2) program elements such as method invocations, package names, and source file names. Having consulted with the relevant



literature [51, 52, 248], we classify the bug reports into three board categories (Steps 1, 2a, 2b and 2c, Fig. 5.1) as follows:

**BR<sub>ST</sub>**: *ST* stands for stack traces. If a bug report contains one or more stack traces besides the regular texts or program elements, it is classified into BR<sub>ST</sub>. Since trace entries contain too much structured information, query generated from such a report is generally considered *noisy*. We apply the following regular expression [163] to locate the trace entries from the report content.

```
(.*)?(.+)\.((.+)\.java:\d+)\|(\(Unknown Source\)|\((Native Method\))
```

**BR<sub>PE</sub>**: *PE* stands for program elements. If a bug report contains one or more program elements (e.g., method invocations, package names, source file name) but no stack traces in the texts, then it is classified into BR<sub>PE</sub>. Queries generated from such report are considered *rich*. We use appropriate regular expressions [211] to identify the program elements from the texts. For example, we use the following one to identify API method invocations within the bug report texts.

```
((\w+)?\.[\s\n\r]*[\w+][\s\n\r]*(?=\(.*\))|([A-Z][a-z0-9+]){2,}
```

**BR<sub>NL</sub>**: *NL* stands for natural language. If a bug report contains neither any program elements nor any stack traces, it is classified into BR<sub>NL</sub>. That is, it contains only unstructured natural language description of the bug. Queries generated from such reports are generally considered *poor* in this work.

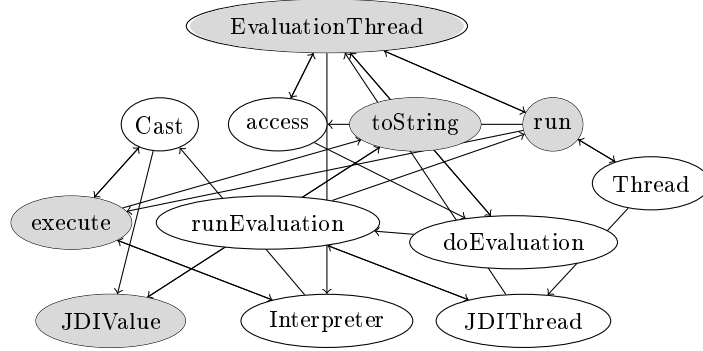
We adopt a semi-automated approach in classifying the bug reports (i.e., the queries). Once a bug report is provided, we employ each of our regular expressions to determine its class. If the automated step fails due to ill-defined structures of the report, the class is determined based on manual analysis. Given the explicit nature of the structured entities, human developers can identify the class easily. The contents of each bug report are considered as the *initial queries* which are reformulated in the next few steps.

## 5.2.2 Query Reformulation

Once bug reports (i.e., queries) are classified into three classes above based on their structured elements or lack thereof, we apply appropriate reformulations to them. In particular, we analyse either bug report contents or the results retrieved by them, employ graph-based term weighting, and then identify important keywords from them for query reformulation as follows:

**Trace Graph Development from BR<sub>ST</sub>**: According to existing findings [193, 248], bug reports containing stack traces are potentially noisy, and performances of the bug localization using such reports (i.e., the queries) are below the average. Hence, important *search keywords* should be *extracted* from the noisy queries for effective bug localization. In this work, we transform the stack traces into a trace graph (e.g., Fig. 5.2) (Steps 3a, 4a, Fig. 5.1, Lines 8–10, Algorithm 6), and identify the important keywords using a graph-based term weighting algorithm namely PageRank [53, 153].

To the best of our knowledge, to date, graph-based term weighting has been employed only on unstructured natural language texts [191] and semi-structured source code [189]. On the contrary, we deal with stack traces



**Figure 5.2:** Trace graph of stack traces in Table 5.1

which are structured and should be analysed carefully. Stack traces generally comprise of an error message containing the encountered exception(s), and an ordered list of method invocation entries. Each invocation entry can be considered as a tuple  $t\{P, C, M\}$  that contains a package name  $P$ , a class name  $C$ , and a method name  $M$ . While these entities are statically connected within a tuple, they are often hierarchically connected (e.g., caller-callee relationships) to other tuples from the traces as well. Hill et al. [104] consider method signatures and field signatures as salient entities from the source code, and suggest keywords from them for code search. Similarly, we consider class name and method name from each of the  $N$  tuples as the salient items, and represent them as the nodes and their dependencies as the connecting edges in the graph. In stack traces, the topmost entry (i.e.,  $i = 1$ ) has the highest degree of interest [70] which gradually decreases for the entries at the lower positions in the list. That is, if  $t_i\{P_i, C_i, M_i\}$  is a tuple under analysis, and  $t_j\{P_j, C_j, M_j\}$  is a neighbouring tuple with greater degree of interest, then the nodes  $V_i$  and edges  $E_i$  are added to the trace graph  $G_{ST}$  as follows:

$$\begin{aligned}
 V_i &= \{C_i, M_i\}, E_i = \{C_i \leftrightarrow M_i\} \cup \{C_i \rightarrow C_j, M_i \rightarrow M_j\} \mid j = i - 1 \\
 V &= \bigcup_{i=1}^N \{V_i\}, E = \bigcup_{i=1}^N \{E_i\}, G_{ST} = (V, E)
 \end{aligned}
 \tag{5.1}$$

For the example stack traces in Table 5.1, the following connecting edges:  $\text{JDIValue} \leftrightarrow \text{toString}$ ,  $\text{Cast} \leftrightarrow \text{execute}$ ,  $\text{Cast} \rightarrow \text{JDIValue}$ ,  $\text{execute} \rightarrow \text{toString}$ ,  $\text{Interpreter} \leftrightarrow \text{execute}$ , and  $\text{Interpreter} \rightarrow \text{Cast}$  are added to the example trace graph in Fig. 5.2.

**Text Graph Development from  $\text{BR}_{PE}$ :** Bug reports containing relevant program entities (e.g., method names) are found effective as queries for IR-based bug localization [193, 220, 248]. However, we believe that *appropriate keyword selection* from such reports can further *boost up* the localization performance. Existing studies employ TextRank and POSRank on natural language texts, and identify search keywords for concept location [191] and information retrieval [53, 153]. Although bug reports (i.e., from  $\text{BR}_{PE}$ ) might contain certain structures such as program entity names (e.g., class name, method name) and code snippets besides natural language texts, the existing techniques could still be applied to them given that these structures are treated appropriately. We thus remove stop words [25] and programming keywords

[26] from a bug report, *split* the structured tokens using *Samurai* (i.e., a state-of-the-art token splitting tool [79]), and then transform the preprocessed report ( $R_{pp}$ ) into a set of sentences ( $S \in R_{pp}$ ). We adopt Rahman and Roy [191] that exploits *co-occurrences* and *syntactic dependencies* among the terms for identifying important terms from a textual body (e.g., change request). We thus develop two text graphs (Steps 3b, 4b, Fig. 5.1, Lines 10–11, Algorithm 6) using co-occurrences and syntactic dependencies among the words from each report as follows:

(1) *Text Graph using Word Co-occurrences*: In natural language texts, the semantics (i.e., senses) of a given word are often determined by its contexts (i.e., surrounding words) [154, 156, 268]. That is, co-occurring words complement the semantics of each other. We thus consider a sliding window of size  $K$  (e.g.,  $K = 2$ ) [153], capture co-occurring words, and then encode the word co-occurrences within each window into connecting edges  $E$  of a text graph [191]. The individual words ( $\forall w_i \in V$ ) are denoted as nodes in the graph. Thus, for a word  $w_i$ , the following node  $V_i$  and two edges  $E_i$  will be added to the text graph  $G_{PE}$  as follows:

$$\begin{aligned} V_i &= \{w_i\}, E_i = \{w_i \leftrightarrow w_{i-1}, w_i \leftrightarrow w_{i+1}\} \mid S = [w_1..w_i..w_N] \\ V &= \bigcup_{\forall S \in R_{pp}} \bigcup_{w_i \in S} \{V_i\}, E = \bigcup_{\forall S \in R_{pp}} \bigcup_{w_i \in S} \{E_i\}, G_{PE} = (V, E) \end{aligned} \quad (5.2)$$

Thus, the example phrase—“*source code directory*”—yields two edges, “*source*”  $\leftrightarrow$  “*code*” and “*code*”  $\leftrightarrow$  “*directory*” while extending the text graph with three distinct nodes— “*source*”, “*code*” and “*directory*”.

(2) *Text Graph using POS Dependencies*: According to *Jespersen’s Rank* theory [53, 113, 191], parts of speech (POS) from a sentence can be divided into three ranks— *primary* (i.e., noun), *secondary* (i.e., verb, adjective) and *tertiary* (i.e., adverb)— where words from a higher rank generally define (i.e., modify) the words from the same or lower ranks. That is, a noun modifies only another noun whereas a verb modifies another noun, verb or an adjective. We determine POS tags using Stanford POS tagger [244], and encode such syntactic dependencies among words into connecting edges and individual words as nodes in a text graph. For example, the sentence annotated using Penn Treebank tags [244]—“*Open*<sub>VB</sub> *the*<sub>DT</sub> *source*<sub>NN</sub> *code*<sub>NN</sub> *directory*<sub>NN</sub>”— has the following syntactic dependencies: “*source*”  $\leftrightarrow$  “*code*”, “*code*”  $\leftrightarrow$  “*directory*”, “*source*”  $\leftrightarrow$  “*directory*”, “*open*”  $\leftarrow$  “*source*”, “*open*”  $\leftarrow$  “*code*” and “*open*”  $\leftarrow$  “*directory*”, and thus adds six edges to the text graph.

**Source Term Graph Development for BR<sub>NL</sub>**: Bug reports containing only natural language texts and no structured entities are found not effective for IR-based bug localization [193, 248]. We believe that such bug reports possibly miss the right keywords for bug localization. Hence, they need to be *complemented* with *appropriate keywords* before using. A recent study [189] provides improved reformulations to a poor natural language query for concept location by first collecting *pseudo-relevance feedback* and then employing graph-based term weighting. In pseudo-relevance feedback, Top-K result documents, returned by a given query, are naively considered as relevant and hence, are selected for query reformulation [62, 98]. Since bug reports from BR<sub>NL</sub> class contain only natural language texts, the above study might directly be applicable to them. We thus adopt their approach for our query reformulation, collect Top-K (e.g.,  $K = 10$ ) source code

documents retrieved by a  $BR_{NL}$ -based query, and develop a source term graph (Steps 3c, 4c, Fig. 5.1, Lines 13–15, Algorithm 6).

Hill et al. [104] consider method signatures and fields signatures from source code as the salient items, and suggest keywords for code search from them. In the same vein, we also collect these signatures from each of the  $K$  feedback documents for query reformulation. In particular, we extract structured tokens from each signature, split them using *Samurai*, and then generate a natural language phrase from each token [104]. For example, the method signature `getContextClassLoader()` can be represented as a verbal phrase – “*get Context Class Loader*”. We then analyse such phrases across all the feedback documents, capture co-occurrences of terms within a fixed window (i.e.,  $K = 2$ ) from each phrase, and develop a source term graph. Thus, the above phrase adds four distinct nodes and three connecting edges – “*get*”  $\leftrightarrow$  “*context*”, “*context*”  $\leftrightarrow$  “*class*” and “*class*”  $\leftrightarrow$  “*loader*” – to the source term graph.

**Term Weighting using PageRank:** Once each body of texts (e.g., stack traces, regular texts, source document) is transformed into a graph, we apply PageRank [57, 153, 189, 191] to the graph for identifying important keywords. PageRank was originally designed for web link analysis, and it determines the reputation of a web page based on the votes or recommendations (i.e., hyperlinks) from other reputed pages on the web [57]. Similarly, in the context of our developed graphs, the algorithm determines importance of a node (i.e., term) based on incoming links from other important nodes of the graph. In particular, it analyses the connectivity (i.e., connected neighbours and their weights) of each term  $V_i$  in the graph recursively, and then calculates the node’s weight  $TW(V_i)$ :

$$TW(V_i) = (1 - \phi) + \phi \sum_{j \in In(V_i)} \frac{TW(V_j)}{|Out(V_j)|} \quad (0 \leq \phi \leq 1) \quad (5.3)$$

Here,  $In(V_i)$  refers to nodes providing incoming links to  $V_i$ ,  $Out(V_j)$  refers to nodes that  $V_j$  is connected to through outgoing links, and  $\phi$  is the damping factor. Brin and Page [57] consider  $\phi$  as the probability of randomly clicking a linked web page and  $1 - \phi$  as the probability of jumping off the page by a random web surfer. They use  $\phi = 0.85$  which was adopted by later studies [53, 153, 191], and we also do the same. We initialize each node in the graph with a value of 0.25 [153], and recursively calculate their weights unless they converge below a certain threshold (i.e., 0.0001) or the iteration count reaches the maximum (i.e., 100) [153]. Once the calculation is over, we end up with an accumulated weight for each node (Step 5, Fig. 5.1, Lines 16–20, Algorithm 6). Such weight of a node is considered as an estimation of relative importance of corresponding term among all the terms (i.e., nodes) from the bug report (i.e., graph).

**Reformulation of the Initial Query:** Once term weights are calculated, we rank the terms based on their weights, and select the Top-K ( $8 \leq K \leq 30$ , Fig. 5.4) terms for query reformulations. Since bug reports (i.e., initial queries) from three classes have different degrees of structured information (or lack thereof), we carefully apply our reformulations to them (Steps 6, 7, Fig. 5.1, Lines 21–30, Algorithm 6). In case of  $BR_{ST}$  (i.e., noisy query), we replace trace entries with the reformulation terms, extract the error message(s)

---

**Algorithm 6** Bug Localization with Query Reformulation and IR

---

```
1: procedure BLIZZARD( $R$ ) ▷  $R$ : a given bug report
2:    $Q' \leftarrow \{\}$  ▷ reformulated query terms
3:   ▷ Classifying and preprocessing the bug report  $R$ 
4:    $C_R \leftarrow \text{getBugReportClass}(R)$ 
5:    $R_{pp} \leftarrow \text{preprocess}(R)$ 
6:   ▷ Representing the bug report as a graph
7:   switch  $C_R$  do
8:     case  $BR_{ST}$ 
9:        $ST \leftarrow \text{getStackTraces}(R)$ 
10:       $G_{ST} \leftarrow \text{getTraceGraph}(ST)$ 
11:     case  $BR_{PE}$ 
12:        $G_{PE} \leftarrow \text{getTextGraphs}(R_{pp})$ 
13:     case  $BR_{NL}$ 
14:        $R_F \leftarrow \text{getPseudoRelevanceFeedback}(R_{pp})$ 
15:        $G_{NL} \leftarrow \text{getSourceTermGraph}(R_F)$ 
16:   ▷ Getting term weights and search keywords
17:   if  $\text{ClassKey } CK \in \{ST, PE, NL\}$  then
18:      $PR_{CK} \leftarrow \text{getPageRank}(G_{CK})$ 
19:      $Q[C_R] \leftarrow \text{getTopKTerm}(\text{sortByWeight}(PR_{CK}))$ 
20:   end if
21:   ▷ Constructing the reformulated query  $Q'$ 
22:   switch  $C_R$  do
23:     case  $BR_{ST}$ 
24:        $N_E \leftarrow \text{getExceptionName}(R)$ 
25:        $M_E \leftarrow \text{getErrorMessage}(R)$ 
26:        $Q' \leftarrow \{N_E \cup M_E \cup Q[C_R]\}$ 
27:     case  $BR_{PE}$ 
28:        $Q' \leftarrow Q[C_R]$ 
29:     case  $BR_{NL}$ 
30:        $Q' \leftarrow \{R_{pp} \cup Q[C_R]\}$ 
31:   ▷ Bug localization with  $Q'$  from codebase  $corpus$ 
32:   return  $\text{Lucene}(corpus, Q')$ 
33: end procedure
```

---

**Table 5.3:** Working Examples

Technique	Group	Query Terms	QE
Baseline	BR <sub>ST</sub>	127 terms from Table 5.1 after preprocessing, <b>Bug ID# 31637, eclipse.jdt.debug</b>	53
BLIZZARD		NullPointerException + “Bug should be able to cast null” + {JDIValue toString execute EvaluationThread run}	<b>01</b>
Baseline	BR <sub>PE</sub>	195 terms (after preprocessing) from <b>Bug ID# 15036, eclipse.jdt.core</b>	27
BLIZZARD		{astvisitor post postvisit previsit pre file post pre astnode visitor}	<b>01</b>
Baseline	BR <sub>NL</sub>	32 terms from Table 5.2 after preprocessing, <b>Bug ID# 187316, eclipse.jdt.ui</b>	30
BLIZZARD		Preprocessed report texts + {compliance create preference add configuration field dialog annotation}	<b>01</b>

QE = Query Effectiveness, rank of the first returned correct result

containing exception name(s), and combine them as the reformulated query. For BR<sub>NL</sub> (i.e., poor query), we combine preprocessed report texts with the highly weighted source code terms as the reformulated query. In the case of BR<sub>PE</sub> category, only Top-K weighted terms from the bug report are used as a reformulated query for bug localization.

### 5.2.3 Bug Localization

**Code Search:** Once a reformulated query is constructed, we submit the query to *Lucene* [98, 164]. Lucene is a widely adopted search engine for document search that combines Boolean search and VSM-based search methodologies (e.g., TF-IDF [114]). In particular, we employ the Okapi BM25 similarity from the engine, use the reformulated query for the code search, and then collect the results (Lines 31–32, Algorithm 6). These resultant and potentially buggy source code documents are then presented as a ranked list to the developer for manual analysis.

**Working Examples:** Table 5.3 shows our reformulated queries for the showcase bug reports in Table 5.1 (i.e., BR<sub>ST</sub>), Table 5.2 (i.e., BR<sub>NL</sub>), and another example report from BR<sub>PE</sub> class. Baseline queries from these reports return their first correct results at the 53<sup>rd</sup> (for BR<sub>ST</sub>), 27<sup>th</sup> (for BR<sub>PE</sub>) and 30<sup>th</sup> (for BR<sub>NL</sub>) positions of their corresponding ranked lists. On the contrary, BLIZZARD refines the noisy query from BR<sub>ST</sub> report, selects important keywords from BR<sub>PE</sub> report, and enriches the poor query from BR<sub>NL</sub> report by adding complementary terms from relevant source code. As a result, all three reformulated queries return their first correct results (i.e., buggy source files) at the topmost (i.e., first) positions, which demonstrate the potential of our technique for bug localization.

**Table 5.4:** Experimental Dataset

System	Time Period	$\mathbf{BR}_{ST}$	$\mathbf{BR}_{PE}$	$\mathbf{BR}_{NL}$	$\mathbf{BR}_{All}$
ecf	Oct, 2001–Jan, 2017	71	319	163	553
eclipse.jdt.core	Oct, 2001–Sep, 2016	159	698	132	989
eclipse.jdt.debug	Oct, 2001–Jan, 2017	126	202	229	557
eclipse.jdt.ui	Oct, 2001–Jun, 2016	130	578	407	1,115
eclipse.pde.ui	Oct, 2001–Jun, 2016	123	239	510	872
tomcat70	Sep, 2001–Aug, 2016	217	731	105	1,053
<b>Total</b>	-	<b>826</b> (16.06%)	<b>2,767</b> (53.81%)	<b>1,546</b> (30.08%)	<b>5,139</b>

$\mathbf{BR}_{ST}$ =Bug reports with stack traces,  $\mathbf{BR}_{PE}$ =Bug reports with program entities but no stack traces,  
 $\mathbf{BR}_{NL}$ =Bug reports with only natural language texts

## 5.3 Experiment

We evaluate our proposed technique in several different dimensions using four widely used performance metrics and more than 5K bug reports (the queries) from six different subject systems. First, we evaluate in terms of the performance metrics and contrast with the baseline for different classes of bug reports/queries (Section 5.3.3). Second, we compare our approach with three state-of-the-art bug localization techniques (Section 5.3.4). Third, and possibly the most importantly, we also compare our approach with four state-of-the-art query reformulations techniques (Section 5.3.4). In particular, we answer four research questions using our experiments as follows:

- **RQ<sub>1</sub>:** (a) How does the proposed approach –BLIZZARD– perform in bug localization, and (b) how do various parameters affect its performance?
- **RQ<sub>2</sub>:** Do our reformulated queries perform better than the baseline search queries from the bug reports?
- **RQ<sub>3</sub>:** Can the proposed approach –BLIZZARD– outperform the existing bug localization techniques including the state-of-the-art?
- **RQ<sub>4</sub>:** Can the proposed approach –BLIZZARD– outperform the existing query reformulation techniques targeting concept/feature location and bug localization?

### 5.3.1 Experimental Dataset

**Dataset Collection:** We collect a total of 5,139 bug reports from six open source subject systems for our experiments. The dataset was taken from an earlier empirical study [193]. Table 5.4 shows our dataset. First, all the resolved (i.e., marked as RESOLVED) bug reports of each subject system were collected from the BugZilla and JIRA repositories given that they were submitted within a specific time interval (Table 5.4). Then the version control history of each system at GitHub was consulted to identify the bug-fixing commits [43]. Such approach was regularly adopted by the relevant literature [49, 163, 276], and we also follow the

same. In order to ensure a fair evaluation, we also discard such bug reports from our dataset for which no source code files (e.g., Java classes) were changed or no relevant source files exist in the collected system snapshot.

**Goldset Development:** We collect *changeset* (i.e., list of changed files) from each of our selected bug-fixing commits, and develop a *goldset*. Multiple changesets for the same bug were merged together.

**Replication Package:** Our working prototype and experimental data are publicly available [27] for replication and reuse.

### 5.3.2 Performance Metrics

We use four performance metrics for the evaluation and comparison of our technique. Since these metrics were frequently used by the relevant literature [163, 191, 220, 249, 268, 276], they are also highly appropriate for our experiments in this work.

**Hit@K:** It is defined as the percentage of queries for which at least one buggy file (i.e., from the goldset) is correctly returned within the Top-K results [250]. It is also called Recall@Top-K [220] and Top-K Accuracy [239] in the literature.

**Mean Average Precision@K (MAP@K):** Unlike regular precision, this metric considers the ranks of correct results within a ranked list. Precision@K calculates precision at the occurrence of each buggy file in the list. Average Precision@K (AP@K) is defined as the average of Precision@K for all the buggy files in a ranked list for a given query [220, 276]. Thus, Mean Average Precision@K is defined as the mean of Average Precision@K (AP@K) of all queries as follows:

$$AP@K = \frac{\sum_{k=1}^D P_k \times buggy(k)}{|S|}, \quad MAP@K = \frac{\sum_{q \in Q} AP@K(q)}{|Q|}$$

Here, function *buggy(k)* determines whether *k<sup>th</sup>* file (or result) is faulty/buggy (i.e., returns 1) or not (i.e., returns 0), and *P<sub>k</sub>* provides the precision at *k<sup>th</sup>* result. *D* refers to the number of total results, *S* is the true positive result set of a query, and *Q* is the set of all queries. The bigger the MAP@K value is, the better a technique is.

**Mean Reciprocal Rank@K (MRR@K):** Reciprocal Rank@K is defined as the multiplicative inverse of the rank of first correctly returned buggy file (i.e., from gold set) within the Top-K results [220, 276]. Thus, Mean Reciprocal Rank@K (MRR@K) averages such measures for all queries in the dataset as follows:

$$MRR@K(Q) = \frac{1}{|Q|} \sum_{q \in Q} \frac{1}{firstRank(q)}$$

Here, *firstRank(q)* provides the rank of first buggy file within a ranked list. MRR@K can take a maximum value of 1 and a minimum value of 0. The bigger the MRR@K value is, the better a bug localization technique is.

**Effectiveness (E):** It approximates a developer’s effort in locating the first buggy file in the result list [98, 163]. That is, the measure returns the rank of first buggy file in the result list. The lower the effectiveness



**Table 5.5:** Performance of BLIZZARD in Bug Localization

Dataset	Technique	Hit@1	Hit@5	Hit@10	MAP@10	MRR@10
BR <sub>ST</sub>	Baseline	21.67%	40.03%	48.25%	28.09%	0.29
	Baseline <sub>Punct</sub>	13.52%	28.25%	37.27%	19.74%	0.20
	BLIZZARD	<b>*34.42%</b>	<b>*66.28%</b>	<b>*75.21%</b>	<b>*45.50%</b>	<b>*0.47</b>
BR <sub>PE</sub>	Baseline	39.85%	64.29%	72.09%	47.28%	0.50
	Baseline <sub>Punct</sub>	25.46%	45.57%	55.39%	32.34%	0.34
	BLIZZARD	<b>44.31%</b>	<b>*69.48%</b>	<b>*77.84%</b>	<b>*52.08%</b>	<b>*0.55</b>
BR <sub>NL</sub>	Baseline	28.24%	50.96%	61.23%	35.48%	0.38
	Baseline <sub>Punct</sub>	21.59%	43.03%	53.17%	28.67%	0.31
	BLIZZARD	<b>29.16%</b>	<b>53.78%</b>	<b>65.21%</b>	<b>*37.62%</b>	<b>0.40</b>
All	Baseline	34.32%	57.83%	66.47%	41.66%	0.44
	Baseline <sub>Punct</sub>	22.56%	42.51%	52.46%	29.55%	0.31
	Baseline <sub>Indri</sub>	32.24%	52.43%	59.51%	39.09%	0.32
	BLIZZARD	<b>*38.58%</b>	<b>*65.08%</b>	<b>*74.52%</b>	<b>*47.13%</b>	<b>*0.50</b>

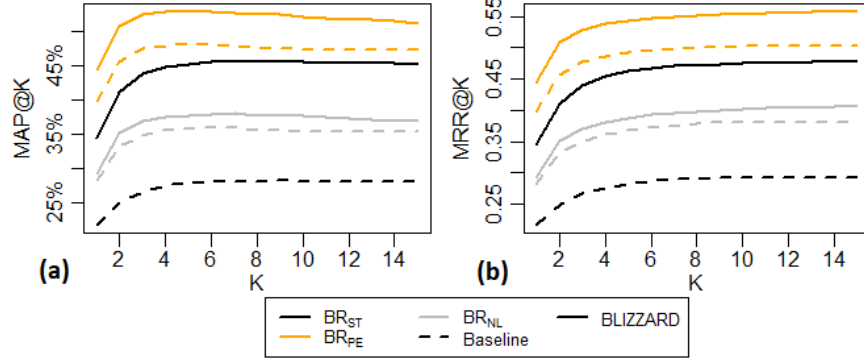
\*=Significantly higher than baseline, **Emboldened**= Comparatively higher

value is, the better a given query is, i.e., the developer needs to check less amount of results from the top before reaching the actual buggy file in the list.

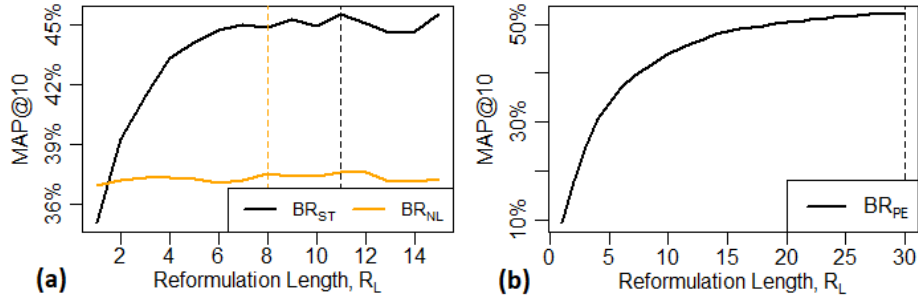
### 5.3.3 Experimental Results

We first show the performance of our technique in terms of appropriate metrics (RQ<sub>1</sub>-(a)), then discuss the impacts of different adopted parameters upon the performance (RQ<sub>1</sub>-(b)), and finally show our comparison with the baseline queries (RQ<sub>2</sub>) as follows:

**Selection of Baseline Queries, and Establishment of Baseline Technique and Baseline Performance:** Existing studies suggest that text retrieval performances could be affected by query quality [98], underlying retrieval engine [164] or even text preprocessing steps [106, 120]. Hence, we choose the baseline queries and baseline technique pragmatically for our experiments. We conduct a detailed study where three independent variables– bug report field (e.g., title, whole texts), retrieval engine (e.g., Lucene [98], Indri [220]) and text preprocessing step (i.e., stemming, no stemming)–are alternated, and then we choose the best performing configuration as the baseline approach. In particular, we chose the preprocessed version (i.e., performed stop word and punctuation removal, split complex tokens but avoided stemming) of the whole texts (i.e., *title + description*) from a bug report as a baseline query. Lucene was selected as the baseline technique since it outperformed Indri on our dataset. The performance of Lucene with the baseline queries was selected as the baseline performance (i.e., Table 5.5) for IR-based bug localization in this study. In short, our baseline is: (preprocessed whole texts + splitting of complex tokens + Lucene search engine).



**Figure 5.3:** Comparison of BLIZZARD with baseline technique in terms of (a) MAP@K and (b) MRR@K



**Figure 5.4:** Impact of query reformulation length on the MAP@10 of our technique—BLIZZARD

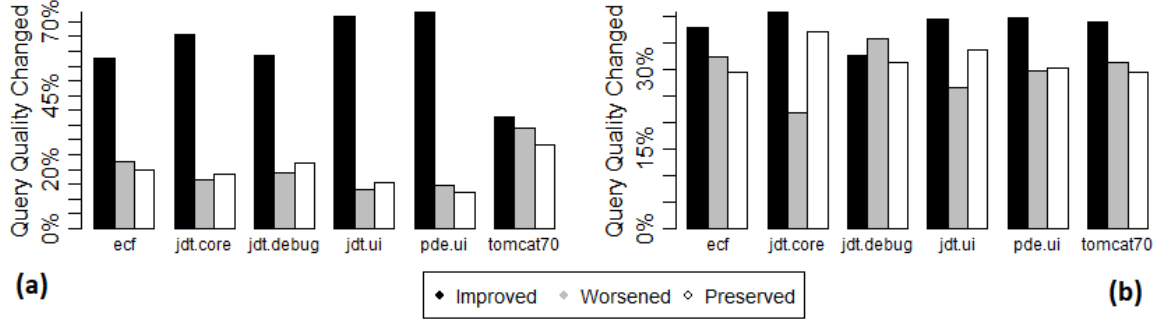
**Answering RQ<sub>1</sub>(a) – Performance of BLIZZARD:** As shown in Table 5.5, on average, our technique—BLIZZARD—localizes 74.52% of the bugs from a dataset of 5,139 bug reports with 47% mean average precision@10 and a mean reciprocal rank@10 of 0.50 which are 12%, 13% and 14% higher respectively than the baseline performance measures. That is, on average, our technique can return the first buggy file at the second position of the ranked list, almost half of returned files are buggy (i.e., true positive) and it succeeds three out of four times in localizing the bugs. Furthermore, while the baseline technique is badly affected by the noisy (i.e.,  $BR_{ST}$ ) and poor queries (i.e.,  $BR_{NL}$ ), our technique overcomes such challenges with appropriate query reformulations, and provides significantly higher performances. For example, the baseline technique can localize 48% of the bugs from  $BR_{ST}$  dataset (i.e., noisy queries) with only 28% precision when Top-10 results are considered. On the contrary, our technique localizes 75% of the bugs with 46% precision in the same context which are 56% and 62% higher respectively than the corresponding baseline measures. Such improvements are about 7% for  $BR_{NL}$ , i.e., poor queries. In the cases where bug reports contain program entities, i.e.,  $BR_{PE}$ , and the baseline performance measures are already pretty high, our technique further refines the query and provides even higher performances. For example, BLIZZARD improves both baseline MRR@10 and baseline MAP@10 for  $BR_{PE}$  dataset by 10% which is promising.

Fig. 5.3 further demonstrates the comparative analyses between BLIZZARD and the baseline technique for various Top-K results in terms of (a) precision and (b) reciprocal rank in the bug localization. From Fig. 5.3-(a), we see that precision reaches to the maximum pretty quickly (i.e., at  $K \approx 4$ ) for both techniques. While

**Table 5.6:** Query Improvement by BLIZZARD over Baseline Queries

Dataset	Query Pair	Improved/MRD	Worsened/MRD	Preserved
BR <sub>ST</sub>	BLIZZARD vs. BL <sub>T</sub>	484 (58.60%)/-82	206 (24.94%)/+34	136 (16.46%)
	BLIZZARD vs. BL	485 (58.72%)/-122	174 (21.07%)/+72	167 (20.22%)
BR <sub>PE</sub>	BLIZZARD vs. BL <sub>T</sub>	1,397 (50.49%)/-60	600 (21.68%)/+38	770 (27.83%)
	BLIZZARD vs. BL	865 (31.26%)/-34	616 (22.26%)/+24	1,286 (46.48%)
BR <sub>NL</sub>	BLIZZARD vs. BL <sub>T</sub>	869 (56.21%)/-27	355 (22.96%)/+29	322 (20.83%)
	BLIZZARD vs. BL	597 (38.62%)/-16	455 (29.43%)/+31	494 (31.95%)
All	BLIZZARD vs. BL <sub>T</sub>	2,750 (53.51%)/-55	1,161 (22.59%)/+32	1,228 (23.90%)
	BLIZZARD vs. BL	1,947 (37.89%)/-50	1,245 (24.22%)/+30	1,947 (37.89)%

**Preserved**=Query quality unchanged, **MRD** = Mean Rank Difference between BLIZZARD and baseline queries, **BL<sub>T</sub>** = title, **BL** = title + description



**Figure 5.5:** Quality improvement of (a) noisy and (b) poor baseline queries by our technique—BLIZZARD

the baseline technique suffers from noisy (i.e., from BR<sub>ST</sub>) and poor (i.e., from BR<sub>NL</sub>) queries, BLIZZARD achieves significantly higher precision than the baseline. Our non-parametric statistical tests—*Mann-Whitney Wilcoxon* and *Cliff’s Delta*—reported *p-values* < 0.05 with a *large* effect size (i.e.,  $0.77 \leq \Delta \leq 1.00$ ). Although the baseline precision for BR<sub>PE</sub> is higher, BLIZZARD offers even higher precision. From Fig. 5.3-(b), we see that mean reciprocal ranks of BLIZZARD have a logarithmic shape and whereas the baseline counterparts look comparatively flat. That is, as more results from the top of the ranked list are considered, more true positives are identified by our technique than the baseline technique does. Statistical tests also reported strong significance (i.e., *p-values* < 0.001) and a *large* effect size (i.e.,  $0.62 \leq \Delta \leq 1.00$ ) of our measures over the baseline counterparts. That is, BLIZZARD performs a good job in reformulating the noisy and poor queries, and such reformulations contribute to a significant improvement in the bug localization performances.

**Answering RQ<sub>1</sub>(b) –Impact of Parameters and Settings:** We investigate the impacts of different adopted parameters -*query reformulation length*, *word stemming*, and *retrieval engine* - upon our technique, and justify our choices. BLIZZARD reformulates a given query (i.e., bug report) for bug localization, and hence, size of the reformulated query is an important parameter. Fig. 5.4 demonstrates how various refor-

mulation lengths can affect the MAP@10 of our technique. We see that precision reaches the maximum for three report classes at different query reformulation lengths (i.e.,  $R_L$ ). For  $BR_{ST}$ , we achieve the maximum precision at  $R_L=11$ , and for  $BR_{NL}$ , such maximum is detected with  $R_L$  ranging between 8 and 12. On the contrary, precision increases in a logarithmic manner for  $BR_{PE}$  bug reports. We investigated up to 30 reformulation terms and found the maximum precision. Given the above empirical findings, we chose  $R_L=11$  for  $BR_{ST}$ ,  $R_L=30$  for  $BR_{PE}$  and  $R_L=8$  for  $R_{NL}$  as the adopted query reformulation lengths and our choices are likely to be justified.

We also investigate the impact of stemming and text retrieval engine on our technique. We found that stemming did not improve the performance of BLIZZARD, i.e., reduced localization accuracy. Similar finding was reported by earlier studies as well [106, 120]. We also found that Lucene performs better than Indri on our dataset. From Table 5.5, we see that Lucene (i.e., **Baseline**) achieves 12% higher Hit@10, 7% higher MAP@10 and 38% higher MRR@10 than those of Indri (i.e., **Baseline<sub>Indri</sub>**). Besides, Lucene has been widely used by the relevant literature [98, 163, 164, 176]. Furthermore, according to a recent third-party study [16], *Apache Lucene* and its variants (e.g., *Solr*, *ElasticSearch*) have  $\approx 77\%$  market share in the enterprise search, which suggests the mass adoption of Lucene in the industrial applications. Given the above findings and earlier suggestions, our choices on stemming and code retrieval engine are also justified.

One might also argue for the inclusion of punctuation marks into the search queries for bug localization. The underlying assumption is that the punctuation marks could provide additional contexts to the search keywords and thus could enrich their semantics which might improve their bug localization performance. However, punctuation marks themselves convey very little semantics compared to the keywords (e.g., natural language terms, identifier names). Thus, the existing literature [120, 220, 250] generally considers them as *noise* and discard them from the analysis. Despite this widely used practice, we investigate the impact of including punctuation marks into the query. In particular, for each subject system, we (1) construct a corpus where source documents are indexed with their terms and punctuation marks, and (2) collect a set of search queries (from the bug reports) that contain both search keywords and punctuation marks. Then we compare between queries with punctuation marks (**Baseline<sub>punct</sub>**) and the same queries without the punctuation marks (**Baseline**). From Table 5.5, we see that the queries with punctuation marks (**Baseline<sub>punct</sub>**) perform poorly compared to their counterpart (**Baseline**) in all measures and in all cases. That is, punctuation marks possibly bring more noise than semantics in the search query. Thus, our choice of discarding punctuation marks from the search query is likely to be justified.

**Summary of RQ<sub>1</sub>:** BLIZZARD outperforms baseline in accuracy, precision and reciprocal rank by 7%–56%, 6%–62% and 6%–62% respectively across three report groups, and our adopted parameters are also justified.

**Answering RQ<sub>2</sub>-Comparison with Baseline Queries:** While Table 5.5 contrasts BLIZZARD with the baseline approach for top 1 to 10 results, we further investigate how BLIZZARD performs compared to the baseline when all results of a query are considered. We compare our queries with two baseline queries

*-title* (i.e.,  $BL_T$ ), *title+description* (i.e.,  $BL$ ) – from each of the bug reports. When our query returns the first correct result at a higher position in the result list than that of corresponding baseline query, we call it *query improvement* and vice versa *query worsening*. When result ranks of the reformulated query and the baseline query are the same, then we call it *query preserving*. From Table 5.6, we see that our applied reformulations improve 59% of the noisy queries (i.e.,  $BR_{ST}$ ) and 39%–56% of the poor (i.e.,  $BR_{NL}$ ) queries both with  $\approx 25\%$  worsening ratios. That is, the improvements are more than two times the worsening ratios. Fig. 5.5 further demonstrates the potential of our reformulations where improvement, worsening and preserving ratios are plotted for each of the six subject systems. We see that noisy queries get benefited greatly from our reformulations, and on average, their query effectiveness improve up to 122 positions (i.e., MRD of  $BR_{ST}$ , Table 5.6) in the result list. Such improvement of ranks can definitely help the developers in locating the buggy files in the result list more easily. The poor queries also improve due to our reformulations significantly (i.e.,  $p\text{-value}=0.004<0.05$ , *Cliff’s*  $\Delta=0.94$  (*large*)), and the correct results can be found 16 positions earlier (than the baseline) in the result list starting from the top. Quantile analysis in Table 5.9 also confirms that noisy and poor queries are significantly improved by our provided reformulations. Besides, the benefits of query reformulations are also demonstrated by our findings in Table 5.5 and Fig. 5.3.

**Summary of RQ<sub>2</sub>:** Our applied reformulations to the bug localization queries improve 59% of the noisy queries and 39%–56% of the poor queries, and return the buggy files closer to the top of result list. Such improvements can reduce a developer’s effort in locating bugs.

### 5.3.4 Comparison with Existing Techniques

**Answering RQ<sub>3</sub> –Comparison with Existing IR-Based Bug Localization Techniques:** Our evaluation of BLIZZARD with four widely used performance metrics shows promising results. The comparison with the best performing baseline shows that our approach outperforms the baselines. However, in order to further gain confidence and to place our work in the literature, we also compared our approach with three IR-based bug localization techniques [220, 250, 276] including the state-of-the-art [250]. Zhou et al. [276] first employ improved Vector Space Model (i.e., rVSM) and bug report similarity for locating buggy source files for a new bug report. Saha et al. [220] employ structured information retrieval where (1) a bug report is divided into two fields–*title*, *description* and a source document is divided into four fields–*class*, *method*, *variable* and *comments*, and then (2) eight similarity measures between these two groups are accumulated to rank the source document. We collect authors’ implementations of both techniques for our experiments.

While the above studies use bug report contents only, the later approaches combine them [221] and add more internal [258] or external information sources such as version control history [249] and author information [250]. In the same vein, Wang and Lo [250] recently combine five internal and external information sources - similar bug report, structured IR, stack traces, version control history and bug reporter’s history – for ranking a source document, and outperform five earlier approaches which makes it the state-of-the-art in

IR-based bug localization. Given that authors' implementation is not publicly available, we implement this technique ourselves by consulting with the original authors. Since BLIZZARD does not incorporate any external information sources, to ensure a fair comparison, we also implement a variant of the state-of-the-art namely AmaLgam+<sub>BRO</sub> where BRO stands for Bug Report Only. It combines bug report texts, structured IR and stack traces (i.e., Table 5.8) for source document ranking.

**Table 5.7:** Comparison with IR-Based Bug Localization Techniques

RG	Technique	Hit@1	Hit@5	Hit@10	MAP@10	MRR@10
BR <sub>ST</sub>	BugLocator	28.79%	55.08%	67.00%	38.49%	0.40
	BLUiR	23.38%	44.34%	54.06%	30.96%	0.32
	AmaLgam+ <sub>BRO</sub>	45.33%	66.97%	73.29%	52.88%	0.55
	<b>BLIZZARD</b>	34.42%	66.28%	<b>75.21%</b>	45.50%	0.47
	<b>BLIZZARD</b> <sub>BRO</sub>	<b>47.42%</b>	<b>73.74%</b>	<b>78.77%</b>	<b>56.22%</b>	<b>0.59</b>
	AmaLgam+	50.51%	66.47%	71.66%	55.97%	0.58
	<b>BLIZZARD</b> +	<b>53.39%</b>	<b>*76.12%</b>	<b>*80.03%</b>	<b>60.65%</b>	<b>0.63</b>
BR <sub>PE</sub>	BugLocator	36.25%	61.37%	70.96%	44.24%	0.47
	BLUiR	35.54%	62.93%	72.17%	43.67%	0.47
	AmaLgam <sub>BRO</sub>	33.90%	60.48%	69.09%	42.00%	0.45
	<b>BLIZZARD</b>	<b>*44.31%</b>	<b>*69.48%</b>	<b>77.84%</b>	<b>*52.08%</b>	<b>*0.55</b>
	<b>BLIZZARD</b> <sub>BRO</sub>	<b>47.16%</b>	<b>71.26%</b>	<b>78.25%</b>	<b>53.69%</b>	<b>0.57</b>
	Amalgam+	52.00%	68.54%	72.93%	55.80%	0.59
	<b>BLIZZARD</b> +	<b>56.84%</b>	<b>74.70%</b>	<b>80.09%</b>	<b>60.78%</b>	<b>0.65</b>
BR <sub>NL</sub>	BugLocator	25.11%	48.52%	59.04%	32.19%	0.35
	BLUiR	29.87%	56.63%	66.10%	38.07%	0.41
	AmaLgam+ <sub>BRO</sub>	29.40%	56.07%	65.01%	37.74%	0.40
	<b>BLIZZARD</b>	<b>29.16%</b>	<b>53.78%</b>	<b>65.21%</b>	<b>37.62%</b>	<b>0.40</b>
	<b>BLIZZARD</b> <sub>BRO</sub>	<b>35.45%</b>	<b>58.75%</b>	<b>69.17%</b>	<b>42.26%</b>	<b>0.46</b>
	AmaLgam+	49.72%	65.42%	71.49%	52.74%	0.57
	<b>BLIZZARD</b> +	47.97%	<b>66.24%</b>	<b>74.49%</b>	52.12%	0.56
All	BugLocator	31.85%	57.37%	67.87%	40.17%	0.43
	BLUiR	32.45%	59.18%	68.65%	40.82%	0.44
	Amalgam+ <sub>BRO</sub>	35.03%	61.32%	69.89%	43.36%	0.46
	<b>BLIZZARD</b>	<b>38.58%</b>	<b>65.08%</b>	<b>74.52%</b>	<b>47.13%</b>	<b>*0.50</b>
	<b>BLIZZARD</b> <sub>BRO</sub>	<b>44.26%</b>	<b>69.15%</b>	<b>76.61%</b>	<b>51.41%</b>	<b>*0.55</b>
	AmaLgam+	52.29%	68.53%	73.58%	56.03%	0.59
	<b>BLIZZARD</b> +	<b>54.78%</b>	<b>73.76%</b>	<b>79.66%</b>	<b>59.32%</b>	<b>0.63</b>

RG=Report Group, BRO=Bug Report Only, \*=Significantly higher

**Table 5.8:** Components behind Existing IR-Based Bug Localization

Technique	Bug Report Only				External Resources			MRR
	BRT	BRS	ST	QR	BRH	VCH	AH	
Baseline	●							0.44
BugLocator	●				●			0.43
BLUiR	●	●						0.44
AmaLgam+ <sub>BRO</sub>	●	●	●					0.46
<b>BLIZZARD</b>	●			●				<b>*0.50</b>
<b>BLIZZARD</b> <sub>BRO</sub>	●	●	●	●				<b>*0.55</b>
AmaLgam+	●	●	●		●	●	●	0.59
<b>BLIZZARD+</b>	●	●	●	●	●	●	●	<b>0.63</b>

**BRT**=Bug Report Texts, **BRS**=Bug Report Structures, **ST**=Stack Traces, **QR**=Query Reformulation, **BRH**=Bug Report History, **VCH**=Version Control History, **AH**=Authoring History, **BRO**=Bug Report Only, ●=Feature used

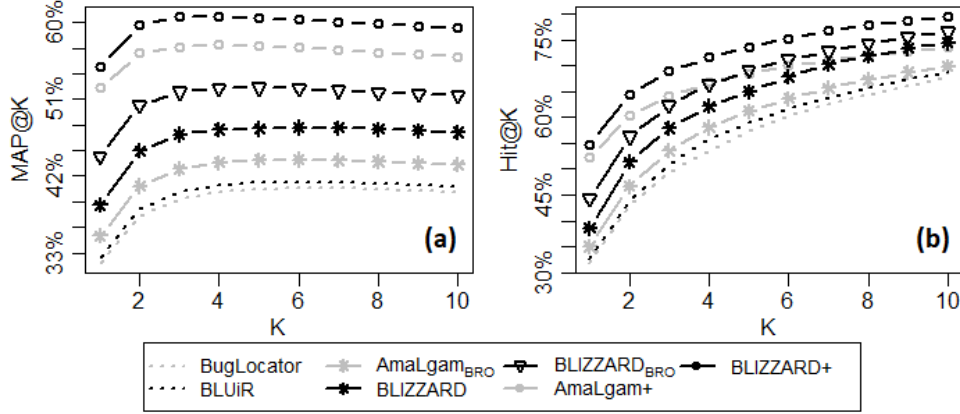
From Table 5.7, we see that AmaLgam+ performs better than the other existing techniques under our study – BugLocator and BLUiR. However, its performance comes at a high cost of mining six information contents (i.e., Table 5.8). Besides, for optimal performance, AmaLgam+ needs past bug reports, version control history and author history which might always not be available. Thus, to ensure a fair comparison, we develop two variants of our technique—BLIZZARD<sub>BRO</sub> and BLIZZARD+. BLIZZARD<sub>BRO</sub> combines query reformulation with bug report only features whereas BLIZZARD+ combines query reformulation with all ranking components of AmaLgam+ (i.e., details in Table 5.8). We then compare both BLIZZARD and BLIZZARD<sub>BRO</sub> with AmaLgam+<sub>BRO</sub>, and BLIZZARD+ with AmaLgam+ respectively.

As shown in Table 5.7, BLIZZARD outperforms AmaLgam+<sub>BRO</sub> in terms of all three metrics especially for BR<sub>PE</sub> reports while performing moderately high with other report groups. For example, BLIZZARD provides 22% higher MRR@10 and 24% higher MAP@10 than AmaLgam+<sub>BRO</sub> for BR<sub>PE</sub>. When all report only features are complemented with appropriate query reformulations, our technique, BLIZZARD<sub>BRO</sub> outperforms AmaLgam+<sub>BRO</sub> in terms of all three metrics—Hit@K, MAP@10 and MRR@10— with each report groups. Such findings suggest that BLIZZARD<sub>BRO</sub> can better exploit the available resources (i.e., bug report contents) than the state-of-the-art variant, and returns the buggy files at relatively higher positions in the ranked list. Furthermore, BLIZZARD+ outperforms the state-of-the-art, AmaLgam+, by introducing query reformulation paradigm. For example, BLIZZARD+ improves Hit@5 and Hit@10 over AmaLgam+ for each of the three query types, e.g., 15% and 12% respectively for noisy queries (BR<sub>ST</sub>). It also should be noted that none of the existing techniques is robust to all three report groups simultaneously. We overcome such issue with appropriate query reformulations, and deliver ≈75%–80% Hit@10 irrespective of the bug report quality. From Table 5.8, we see that BLIZZARD<sub>BRO</sub> provides 20% higher MRR@10 than AmaLgam+<sub>BRO</sub>

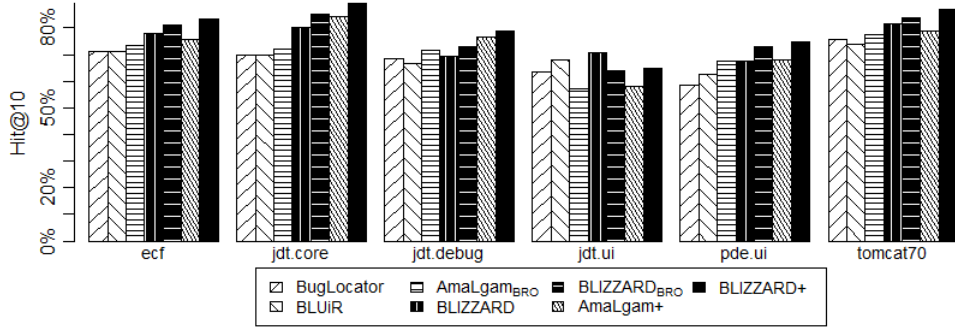
**Table 5.9:** Comparison of Query Effectiveness with Existing Query Reformulation Techniques

Technique	RG	Improvement							Worsening							Preserving
		#Improved	Mean	Q1	Q2	Q3	Min.	Max.	#Worsened	Mean	Q1	Q2	Q3	Min.	Max.	#Preserved
Rocchio [213]		337 (40.80%)	68	4	12	60	1	1,245	264 (31.96%)	118	6	21	97	2	2,824	225 (27.24%)
RSV [212]		218 (26.39%)	163	10	43	158	1	2,103	236 (28.57%)	198	17	71	245	2	2,487	372 (45.04%)
Sisman and Kak [231]	BR <sub>ST</sub>	339 (41.04%)	66	4	12	53	1	1,245	265 (32.08%)	121	7	23	100	2	2,846	222 (26.88%)
STRICT [191]	(826)	399 (48.30%)	35	1	4	17	1	1,538	318 (38.50%)	139	6	25	110	2	3,066	109 (13.20%)
Baseline			153	7	35	149	2	2,221		70	1	5	30	1	2,469	
<b>BLIZZARD</b>		<b>485 (58.72%)</b>	<b>22</b>	<b>1</b>	<b>3</b>	<b>9</b>	<b>1</b>	<b>932</b>	<b>174 (21.07%)</b>	<b>112</b>	<b>4</b>	<b>15</b>	<b>60</b>	<b>2</b>	<b>3,258</b>	<b>167 (20.22%)</b>
Rocchio [213]		32 (2.07%)	33	4	8	19	1	365	24 (1.55%)	140	4	12	146	2	850	1,490 (96.38%)
RSV [212]		345 (22.27%)	112	3	9	38	1	6,564	751 (48.57%)	105	7	23	81	2	2,140	450 (29.11%)
Sisman and Kak [231]	BR <sub>NL</sub>	499 (32.28%)	59	2	6	26	1	2,019	575 (37.19%)	98	5	15	64	2	2,204	472 (30.47%)
STRICT [191]	(1,546)	467 (30.21%)	57	2	6	30	1	1,213	654 (42.30%)	112	5	18	63	2	4,933	425 (27.44%)
Baseline			91	5	15	57	2	2,434		61	2	8	30	1	1,894	
<b>BLIZZARD</b>		<b>597 (38.62%)</b>	<b>75</b>	<b>2</b>	<b>8</b>	<b>32</b>	<b>1</b>	<b>3,063</b>	<b>455 (29.43%)</b>	<b>92</b>	<b>5</b>	<b>15</b>	<b>54</b>	<b>2</b>	<b>2,024</b>	<b>494 (31.95%)</b>





**Figure 5.6:** Comparison of (a) MAP@K and (b) Hit@K with the state-of-the-art IR-based bug localization techniques

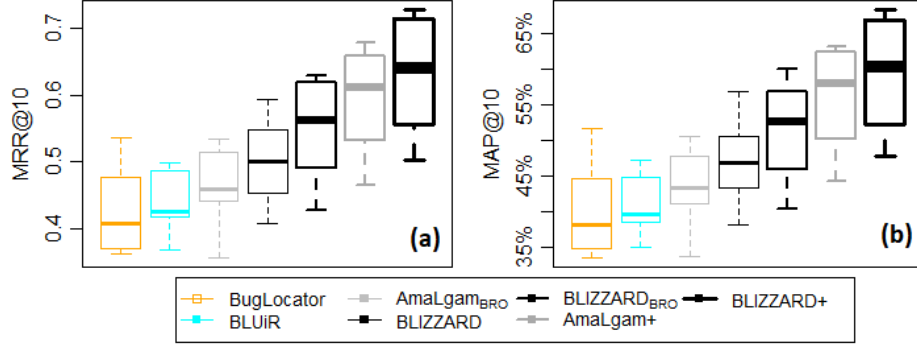


**Figure 5.7:** Comparison of Hit@10 across all subject systems

by consuming equal amount of resources, i.e., bug report only. All these findings above suggest two important points. First, earlier studies might have failed to exploit the report contents and structures properly for bug localization. Second, query reformulation has a high potential for improving the IR-based bug localization.

Fig. 5.6 demonstrates a comparison of BLIZZARD with the existing techniques in terms of (a) MAP@K and (b) Hit@K for various Top-K results. Our statistical tests report that BLIZZARD, BLIZZARD<sub>BRO</sub> and BLIZZARD+ outperform AmaLgam+<sub>BRO</sub> and AmaLgam+ respectively in MAP@K by a significant margin (i.e.,  $p\text{-values} \leq 0.001$ ) and *large* effect size (i.e.,  $0.82 \leq \Delta \leq 1.00$ ). Similar findings were also achieved for Hit@K.

Fig. 5.7 and Fig. 5.8 focus on subject system specific performances. From Fig. 5.7, we see that BLIZZARD outperforms AmaLgam+<sub>BRO</sub> with four systems in Hit@10, and falls short with two systems. However, BLIZZARD<sub>BRO</sub> and BLIZZARD+ outperform AmaLgam+<sub>BRO</sub> and AmaLgam+ respectively for all six systems. As shown in the box plots of Fig. 5.8, BLIZZARD has a higher median in MRR@10 and MAP@10 than AmaLgam+<sub>BRO</sub> across all subject systems. AmaLgam+ improves both measures especially MAP@10. However, BLIZZARD+ provides even higher MRR@10 and MAP@10 than any of the existing techniques including the state-of-the-art.



**Figure 5.8:** Comparison of (a) MRR@10 and (b) MAP@10 with existing techniques across the subject systems

**Summary of RQ<sub>3</sub>:** Our technique outperforms the state-of-the-art from IR-based bug localization in various dimensions. It offers 20% higher precision and reciprocal rank than that of state-of-the-art variant (i.e., AmaLgam+<sub>BRO</sub>) by using only query reformulation rather than costly alternatives, e.g., mining of version control history

**Answering RQ<sub>4</sub> –Comparison with Existing Query Reformulation Techniques:** While we have already showed that our approach outperforms the baselines and the state-of-the-art IR-based bug localization approaches, we also wanted to further evaluate our approach in the context of query reformulation. We thus compared BLIZZARD with four query reformulation techniques [98, 191, 213, 231] including the state-of-the-art [191] that were mostly used for concept/feature location. We use authors’ implementation of the state-of-the-art, STRICT, and re-implement the remaining three techniques. We collect Query Effectiveness (i.e., rank of the first correct result) of each of the reformulated queries provided by each technique, and compare with ours using quantile analysis. From Table 5.9, we see that 48% of the noisy (i.e., BR<sub>ST</sub>) queries are improved by STRICT, and 32% of the poor (i.e., BR<sub>NL</sub>) queries are improved by Sisman and Kak [231]. Neither of these techniques considers bug report quality (i.e., prevalence of structured information or lack thereof) and each technique applies the same reformulation strategy to all reports. On the contrary, BLIZZARD chooses appropriate reformulation based on the class of a bug report, and improves 59% of the noisy queries and 39% of the poor queries which are 22% and 20% higher respectively. When compared using quantile analysis, we see that our quantiles are highly promising compared to the baseline. Our reformulations clearly improve the noisy queries, and 75% of the improved queries return their first correct results within Top-9 (i.e., Q<sub>3</sub>=9) positions whereas STRICT needs Top-17 positions for the same. In the case of poor queries, quantiles of BLIZZARD are comparable to that of Sisman and Kak. However, BLIZZARD worsens less and preserves higher amount of the baseline queries which demonstrate its high potential.

**Summary of RQ<sub>4</sub>:** Our approach, BLIZZARD, outperforms the state-of-the-art in query reformulation using context-aware (i.e., responsive to report quality) query reformulation. Whatever improvements are

offered to noisy and poor queries by the state-of-the-art, our technique improves 22% more of noisy queries and 20% more of the poor queries.

## 5.4 Threats to Validity

Threats to internal validity relate to experimental errors and biases [272]. Replication of existing studies and misclassification of the bug reports are possible sources of such threats. We use authors' implementation of three techniques [191, 220, 276] and re-implement the remaining four. While we cannot rule out the possibility of any implementation errors, we re-implemented them by consulting with the original authors [250] and their reported settings and parameters [98, 213, 231]. While our technique employs appropriate regular expressions for bug report classification, they are limited in certain contexts (e.g., ill-structured stack traces) which require limited manual analysis currently. More sophisticated classification approaches [173, 241, 277] could be applied in the future work.

Threats to external validity relate to generalizability of a technique [272]. We conduct experiments using Java systems. However, since we deal with mostly structured items (e.g., stack traces, program entities) from a bug report, our technique can be adapted to other OOP-based systems that have such structured items.

## 5.5 Related Work

**Bug Localization:** Automated bug localization has been an active research area for over two decades [220]. Existing studies from the literature can be roughly categorized into two broad families—*spectra* based and *information retrieval (IR)* based [130, 248]. We deal with IR-based bug localization in this work. Given that spectra based techniques are costly and lack scalability [163, 248], several studies adopt IR-based methods such as Latent Semantic Indexing (LSI) [179], Latent Dirichlet Allocation (LDA) [167, 207] and Vector Space Model (VSM) [122, 163, 220, 230, 258, 276] for bug localization. They leverage the shared vocabulary between bug reports and source code entities for bug localization. Unfortunately, as existing evidences [193, 248] suggest, they are inherently subject to the quality of bug reports. A number of recent studies complement traditional IR-based localization with spectra based analysis [130], machine learning [128, 269] and mining of various repositories— bug report history [221], version control history [230, 249], code change history [255, 270] and bug reporter history [250]. Recently, Wang and Lo [250] combine bug report contents and three external repositories, and outperform five earlier IR-based bug localization techniques [220, 221, 230, 249, 258, 276] which makes it the state-of-the-art. In short, the contemporary studies advocate for combining (1) multiple localization approaches (e.g., dynamic trace analysis [130], Deep learning [128], learning to rank [268, 269]) and (2) multiple external information sources with classic IR-based localization, and thus, improve the localization performances. However, such solutions could be costly (i.e., multiple repository mining) and less scalable (i.e., dependency on external information sources), and hence, could be infeasible to use in

practice. In this work, we approach the problem differently, and focus on better leveraging the potential of the resources at hand (i.e., bug report and source code) which might have been *underestimated* by the earlier studies. In particular, we refine the noisy queries (i.e., containing stack traces) and complement the poor queries (i.e., lacks structured items), and offer an effective information retrieval unlike the earlier studies. Thus, issues raised by low quality bug reports [248] have been significantly addressed by our technique, and our experimental findings support such conjecture. We compare with three existing studies including the state-of-the-art [250], and the detailed comparison can be found in Section 5.3.4 (i.e., RQ<sub>3</sub>).

A few studies [163, 258] analyse stack traces from a bug report for bug localization. However, they apply the trace entries to boost up source document ranking, and superfluous trace entries were not discarded from their stack traces. Learning-to-rank [268, 269] and Deep learning [128] based approaches might also suffer from noisy and poor queries since they adopt classic IR without query reformulation in their document ranking. Recent studies [245, 268] employ distributional semantics of words to address limitations of VSM. Since noisy terms in the report could be an issue, our approach can complement these approaches through query reformulation.

**Query Reformulation:** There exist several studies [64, 84, 96, 98, 104, 120, 188, 191, 208, 268] that support concept/feature/concern location tasks using query reformulation. However, these approaches mostly deal with unstructured natural language texts. Thus, they might not perform well with bug reports containing excessive structured information (e.g., stack traces), and our experimental findings also support this conjecture (Table 5.9). Sisman and Kak [231] first introduce query reformulation in the context of IR-based bug localization. However, their approach cannot remove noise from a query. Recently, Chaparro et al. [65] identify observed behaviour (OB), expected behaviour (EB) and steps to reproduce (S2R) from a bug report, and then use OB texts as a reformulated query for bug localization. However, they only analyse unstructured texts whereas we deal with both structured and unstructured contents. Since we apply query reformulation, we compare with four recent query reformulation techniques employed for concept location–Rocchio [213], RSV [212], STRICT [191] [189] and bug localization–SCP [231]. The detailed comparison can be found in Section 5.3.4 (i.e., RQ<sub>4</sub>).

In short, existing IR-based techniques suffer from *quality issues* of bug reports whereas traditional query reformulation techniques are *not well-adapted* for the bug reports containing excessive structured information (e.g., stack traces). Our work fills this *gap* of the literature by incorporating context-aware (i.e., report quality aware) query reformulation into the IR-based bug localization. Our technique better exploits resources at hand and delivers equal or higher performance than the state-of-the-art at a relatively lower cost. To the best of our knowledge, such comprehensive solution was not provided by any of the existing studies.

## 5.6 Summary

Developers spend about 50% of their development time in dealing with software bugs and failures, which cost billions of dollars every year [1]. Finding the locations of bugs within the source code is a crucial step

of the bug resolution process. Traditional solutions for bug localization are limited. They do not perform well when the bug reports are noisy or poor in quality [248]. In this chapter, we propose a novel technique –BLIZZARD– that accepts a bug report as a search query, employs appropriate query reformulations based on its reporting quality (e.g., noisy, poor), and then delivers an improved, reformulated search query for the bug localization. Experiments using 5,139 bug reports from six open source subject systems report that BLIZZARD can offer up to 62% higher precision than the best baseline and 20% higher precision than the state-of-the-art measure. Our technique also improves 22% more of noisy queries and 20% more of the poor queries than that of the state-of-the-art.

Although our approach suggests appropriate search queries from the noisy and poor quality bug reports, they might require further reformulations to achieve the optimal performance during bug localization. In particular, we notice that BLIZZARD might achieve only marginal improvement over the baseline when the bug reports are really poor. In the next chapter, our fourth study (BLADER, Chapter 6) attempts to overcome this challenge. BLADER accepts a poor bug report as a search query, and reformulates the query using word embedding technology and clustering tendency analysis for the bug localization.

# CHAPTER 6

## SEARCH QUERY REFORMULATION FOR BUG LOCALIZATION USING WORD SEMANTICS & CLUSTERING TENDENCY ANALYSIS

Software bugs and failures are pervasive in modern software systems [1]. Changes to the existing software systems are also frequent and inevitable during the maintenance. Thus, finding bugs in the software code (a.k.a., bug localization) and identifying concepts of interest in the software code (a.k.a., concept location) are two major challenges of the software maintenance. Our previous studies (Chapters 3, 4, 5) deliver search queries for concept location and bug localization by analysing change requests, bug reports and relevant source code documents. They determine keyword importance using statistical properties (e.g., term co-occurrences, Section 3.3.3) and dependency relationships among the keywords (e.g., syntactic, static or hierarchical dependencies, Section 5.2.2). While these dimensions were found promising, the underlying semantics of the keywords were overlooked, which could have been another important dimension. In this chapter, we overcome this issue with another study. Here, we present BLADER that accepts a poor quality bug report as a query, analyses clustering tendency between the query and the candidate keywords in terms of their underlying semantics, and then delivers an improved, reformulated search query for the bug localization.

The rest of the chapter is organized as follows – Section 6.1 presents an overview of our study, and Section 6.2 offers a motivating example. Section 6.3 describes our proposed technique for search query reformulation for bug localization, and Section 6.4 discusses our experiments, evaluation and validation. Section 6.5 identifies the threats to the validity of our findings, Section 6.6 discusses related work, and finally Section 6.7 concludes the chapter with future work.

### 6.1 Introduction

Software bugs and failures cost trillions of dollars every year and consume almost half of the development time and efforts [1, 28]. Bug localization is one of the most challenging steps of software debugging. It involves finding out the bugs or faults within the source code of a software system [130, 170]. Over the last two decades, Information Retrieval (hereby **IR**) had been widely adopted in the localization of software bugs [98, 163, 164, 220, 249, 276]. IR-based localization leverages *textual similarity* between a bug report (query)

and the source code in localizing the bug [276]. Such a localization is reported as light-weight, cost-effective and even as accurate as spectra-based techniques which localize a bug by analysing the execution traces [207, 248]. Unfortunately, recent findings [192, 193, 248] suggest that IR-based approaches suffer heavily when the quality of a bug report is low (i.e., poor query). They cannot perform well without the presence of relevant program entity names (e.g., class names, method names) in the report texts. Such entities could essentially help find the locations of the encountered bugs or failures within the source code. According to existing investigations [193, 248], up to **55%** of bug reports of a software system could be of low quality (i.e., lack program entities). Thus, such automated supports are highly warranted that could localize the bugs in the software even with these low quality bug reports (i.e., poor queries).

Software developers often use a few important keywords from a bug report as a *search query* for bug localization [120, 231]. Unfortunately, choosing such keywords from the bug report is often challenging, and even the experienced developers cannot do this job well [83, 120, 142]. Furthermore, they might not even find any useful keywords from the low quality bug reports. An ad hoc alternative to this issue could be the use of the whole texts (i.e., *title + description*) of a bug report as a search query. Unfortunately, such texts might also produce verbose and poor queries [64]. Thus, appropriate query selection for bug localization is a major challenge, and the developers are badly in need of automated tool supports. Our work in this chapter addresses this particular research problem— **query reformulation** for the IR-based bug localization.

Several existing studies offer automated supports for query reformulations in the context of concept/feature location [84, 98, 104, 164, 191] and bug localization [65, 163, 192, 231]. Unfortunately, they might fail with low quality bug reports (poor queries) due to their high reliance on the report contents. Most of them use pseudo-relevance feedback (PRF) [222] and term selection methods (e.g., TF-IDF [114]). PRF-based techniques accept a search query, retrieve a few apparently relevant source documents, and then expand the query with important keywords extracted from these documents [222]. However, if the given query is already poor, the retrieved documents could be either noisy or even completely irrelevant which can negatively affect the reformulated query. Low quality bug reports (i.e., poor queries) generally do not contain any relevant program entity names (e.g., class names) [193, 248]. Thus, term selection based approaches [120, 191] might also not be sufficient enough for selecting appropriate queries from the low quality bug reports.

In this chapter, we propose and design a novel technique—**BLADER**— that reformulates a poor search query by analysing and complementing its underlying semantics, and then localizes the bugs in the software code using the reformulated query. First, we construct a high dimensional semantic space (a.k.a., *semantic hyperspace*) and a large vocabulary of  $\approx$ **660K** words by employing a widely used text mining tool, FastText [54], on **1.40 million** Q&A threads of Stack Overflow. FastText represents each word of the vocabulary as a *point* within the semantic hyperspace so that similar or relevant words *cluster* together within the space. Second, we collect multiple reformulation candidates from the relevant source code against a given query, and determine their appropriateness based on their *clustering tendency* towards the given query within the hyperspace (i.e., use of *word semantics*). Third, we choose the best candidate using machine learning as our

**Table 6.1:** An Example of Low Quality Bug Report (Issue #192756, ECF)

Field	Content
Title	[IRC] On channel join, get rid of 'entered' spam in
Description	If you join a big channel, you get a ton of "xxx entered". I think on channel entry, we don't show these messages. We should show these messages in maybe the 'server tab', ie., irc.freenode.net, similar to how other IRC clients do it.

#### An Example of Query Reformulation

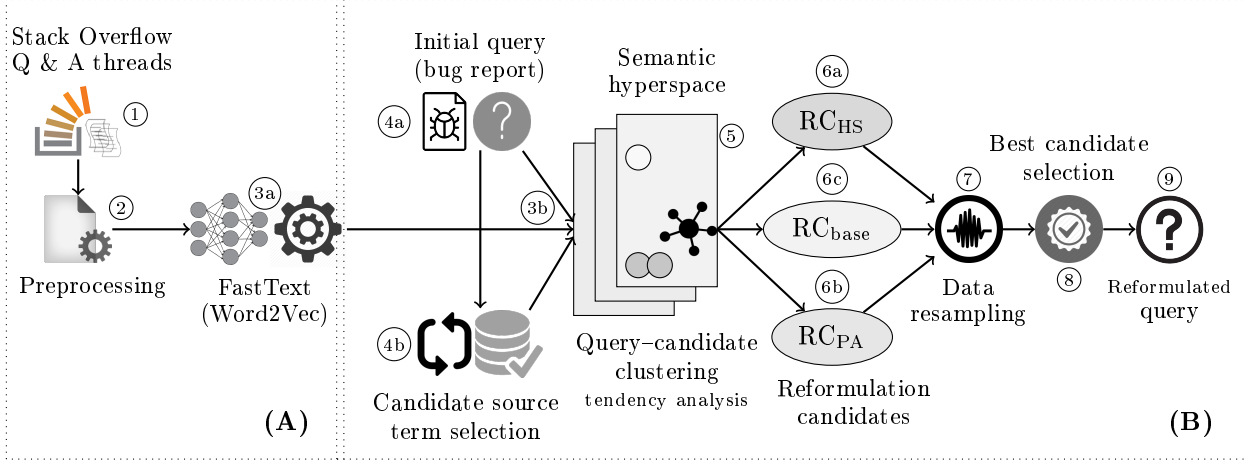
Technique	Reformulated Query	QE
Baseline <sub>T</sub>	{title}	30
Baseline <sub>D</sub>	{description}	10
Baseline	{title + description}	12
Rocchio [213]	{title + description} + {remoteserviceadminevent admin service feed remote synd mask writer event export}	10
<b>BLADER</b> (Proposed)	{title + description} + {connect invitation handle message room chat user send}	<b>03</b>

QE = Rank of the first buggy document retrieved by the query

reformulated query (i.e., *poor query + complementary keywords*), and then employ this query in the bug localization with information retrieval. While many earlier approaches [220, 276] simply use the poor queries (i.e., low quality bug reports) verbatim, our approach complements such queries with relevant keywords from the source code using *word semantics*, *clustering tendency analysis* and *machine learning*. Thus, our approach, BLADER, has a *greater* chance of overcoming the challenges posed by low quality bug reports (poor queries). To the best of our knowledge, this is the first attempt to apply *clustering tendency analysis* and *word semantics* (derived from Stack Overflow Q&A threads) in the query reformulation intended for bug localization, which makes our work *novel*.

We evaluate our technique using *four* widely used performance metrics, *two* different dimensions (bug localization and query reformulation), and a dataset of **1,546** bug reports from *six* subject systems. First, we evaluate in terms of bug localization performance (**RQ<sub>1</sub>**). In contrast with a baseline approach, our technique localizes software bugs with **9%** higher Hit@10, **17%** higher MAP and **21%** higher MRR. Second, we compare with *five* well known existing studies on bug localization [192, 220, 250, 268, 276] (**RQ<sub>3</sub>**), where BLADER achieves **11%** higher MAP and **15%** higher MRR than the state-of-the-art [192]. Third, we compare with *six* well known existing approaches on query reformulation [188, 191, 192, 212, 213, 231] (**RQ<sub>4</sub>**). BLADER improves **48%-72%** of the poor queries which is **23%** higher than that of the state-of-the-art [192]. Such an improvement can help the **practitioners** find out the buggy documents with **less manual or cognitive efforts**. Our work also demonstrates the *novel* and *effective* application of *clustering tendency analysis* and *word semantics* derived from Stack Overflow in addressing a complex Software Engineering challenge such as *query reformulation for IR-based bug localization*. Thus, our work makes the following contributions.





**Figure 6.1:** Schematic diagram of the proposed query reformulation technique –BLADER– (A) Construction of a semantic hyperspace and (B) Reformulation of a query for bug localization

- (a) A novel query reformulation technique, **BLADER<sub>QR</sub>**, that improves a poor search query with appropriate keywords from the source code of a software system by employing *word semantics*, *clustering tendency analysis*, machine learning and pseudo-relevance feedback.
- (b) A novel bug localization technique, **BLADER<sub>BL</sub>**, that employs query reformulation, Information Retrieval, and word semantics for the bug localization.
- (c) Comprehensive evaluation of the proposed approach using **1,546** bug reports and comparison with **eleven** existing techniques in **two** different dimensions.
- (d) A replication package [29] that includes a working prototype, experimental dataset and other associated materials for third party reuse and replication.

## 6.2 Motivating Example

In order to demonstrate the capability of our technique in improving search queries and thereby in bug localization, we provide an example where BLADER outperforms three baseline approaches and one frequently cited existing approach–Rocchio [213]. The upper part of Table 6.1 shows a low quality bug report. The report explains a bug on *IRC chat* using only regular texts. It does not contain any program entities (e.g., class names) which might have assisted in the bug localization process. In the lower part of Table 6.1, we compare our approach with others. We see that three baseline approaches – Baseline<sub>T</sub> (*title* only), Baseline<sub>D</sub> (*description* only) and Baseline (*title+description*) do not perform well, and they return the first buggy document at the 30<sup>th</sup>, 10<sup>th</sup> and 12<sup>th</sup> positions respectively in their result lists. On the contrary, BLADER **complements** this low quality bug report (poor query) with **appropriate search keywords** from the **source code**, and then returns the same result at the **third** position which is a **70%** rank improvement over the baseline. Despite the expansion, Rocchio’s method fails to improve over baseline due to its high reliance on the poor query. Please note that our keywords are **more appropriate and more relevant**.

## 6.3 BLADER: Automated Query Reformulation using Word Semantics & Clustering Tendency Analysis for Bug Localization

Fig. 6.1 shows the schematic diagram of our proposed technique for query reformulation targeting bug localization. Furthermore, Algorithm 7 shows the pseudo code of our approach. We first construct a *semantic hyperspace* (i.e., multi-dimensional semantic space) from 1.40 million Q&A threads of Stack Overflow using FastText. Then we reformulate a poor query using two clustering tendency metrics derived from this hyperspace and perform bug localization with the reformulated query as follows:

### 6.3.1 Construction of a Semantic Hyperspace from Stack Overflow Q&A Threads

Several earlier studies [144, 226] make use of natural language thesauri (e.g., WordNet) to expand a search query with synonyms and semantically similar words. However, Sridhara et al. [233] later suggest that words used in Software Engineering literature have different semantics than what they have in the regular texts (e.g., news article). That is, natural language thesaurus such as WordNet might not be sufficient enough for keyword suggestion in the context of Software Engineering (SE) tasks (e.g., bug localization). We thus construct a dictionary-like mechanism (e.g., *semantic hyperspace*) using the contents relevant to Software Engineering (e.g., programming Q & A threads) for our query reformulation.

**Data Collection:** We use Stack Overflow, the most popular programming Q & A site on the web, for our semantic hyperspace construction (i.e., Step 1, Fig. 6.1). Stack Overflow is a large body of knowledge with 14 million questions and 22 million answers across various programming languages, algorithms, API libraries, and state-of-the-art software development practices [58]. These contents are spontaneously produced by millions of software engineers, programming hobbyists, and researchers from all over the world. They are also systematically curated by this large technical crowd with a voting based system. Thus, Stack Overflow offers a massive body of relevant, reusable, and useful resources. Furthermore, such resources could be leveraged in exploring the *underlying semantics* of the keywords from queries related to Software Engineering tasks. Several earlier studies [58, 201, 205, 272] also demonstrate the high potential of Stack Overflow for SE tasks.

We collect a total of 1.40 million questions and answers related to *Java* from Stack Overflow for our corpus preparation. We make use of the official data dump [35] released on March, 2018 by Stack Overflow. Since we deal with Java-based subject systems, we were interested in the Java related questions, answers and discussions posted on Stack Overflow. We identify them using <java> tag of the questions. We also make sure that each of the questions has at least one answer in order to avoid the low quality questions.

**Text Preprocessing:** As a widely used practice [272], we perform standard natural language preprocessing on each of the questions and answers (i.e., Step 2, Fig. 6.1). In particular, we first remove stop words, programming keywords, punctuation marks, and digits, and then split the complex or structured tokens (e.g., camel case, `GoogleTalk`) into simpler ones (e.g., `Google`, `Talk`). Stop words (e.g., ‘a’, ‘an’, ‘the’)

and keywords (e.g., `for`, `while`) are frequently used words in the texts and source code respectively which convey very little semantics. We use a standard list of stop words [25] and an official list of Java language keywords [12] for the stop word removal and keyword removal respectively. It should be noted that we avoid stemming in our analysis due to the controversial evidence of stemming in the software text retrieval [106].

**Learning of Word Embeddings:** Semantics of a word are often determined by its contexts (i.e., surrounding words) within the texts. The same word can express different meanings in different contexts. For example, the word ‘*bank*’ could mean a financial institution in one context, and could also mean an edge of the river in another context. Thus, determining the *exact meaning* of a word is a major challenge. Since bug reports are written by mostly *layman* users, search queries constructed from these reports could be ambiguous as well. There have been several attempts [109, 154, 188, 233, 265, 272] for understanding software word semantics during the last decade. Recently, Mikolov et al. [156] propose a neural network based approach called *Word2Vec* that learns the semantics of a word in terms of a numeric vector. Such a vector is also called *word embeddings*. Word2Vec has been found surprisingly effective for various traditional text classification tasks (e.g., sentiment analysis [115, 238]). We use a recent version of their approach namely *FastText* [54] for learning the word embeddings in our work.

We use a three-layer neural network (i.e., input layer + hidden layer + output layer) with Skip-gram algorithm to learn word embeddings (Step 3a, Fig. 6.1). The training of this network (1) starts with a set of random weights and an activation function (e.g., linear function) in the hidden layer, and (2) finishes with a set of fine-tuned weights in the hidden layer neurons.

Then the hidden layer weights learned by the network for each of the words are considered as their corresponding *word embeddings* [36, 156]. We implement FastText using `gensim` library in Python platform, and use the default set of parameters like earlier studies [194, 274]. In particular, we use a *window size* of 5, a *minimum frequency* of 5, and a hidden layer of 100 neurons. Thus, our trained model returns a numeric vector of 100 weights as the *word embeddings* for each of the  $\approx 660\text{K}$  words from the corpus.

**Modelling of a Semantic Hyperspace with Word Embeddings:** Although word embeddings are learned from a simple, shallow neural network, they have useful properties that could be leveraged for various text processing tasks (e.g., semantic similarity estimation [268, 274]). For example, the embeddings are learned in such a fashion that similar or relevant words are found close to one another when their word embeddings are visualized. We make use of this interesting property, and map the embedding vector of each word to a unique *point* within the *semantic hyperspace*. *Hyperspace* refers to a space having more than three dimensions. Thus, each point (or word) could be considered as an *intersection* of multiple dimensions pointing towards *multiple semantics*. Our corpus from Stack Overflow contains a total of  $\approx 660\text{K}$  unique words, and their word embeddings thus construct a large-scale, multi-dimensional semantic space (a.k.a., semantic hyperspace) (Step 3b, Fig. 6.1).

---

**Algorithm 7** Bug Localization with QR using Word Semantics & Clustering Tendency Analysis

---

```
1: procedure BLADER( $Q, WE$ )
2:    $\triangleright Q$ : a poor bug report, a.k.a., search query
3:    $\triangleright WE$ : word embeddings learned from Stack Overflow
4:    $\triangleright$  stopword/keyword/punctuation removal, and token splitting
5:    $Q_{pp} \leftarrow \text{preprocess}(Q)$ 
6:    $\triangleright$  collect candidate terms for reformulation
7:    $C \leftarrow \text{getCandidateTermsFromProjectSource}(Q_{pp})$ 
8:    $X \leftarrow \{C \cup Q_{pp}\}$ 
9:    $\triangleright$  get reformulation candidates using clustering tendency
10:   $S \leftarrow \{\forall x : x \in X\}, S' \leftarrow \{\forall x : x \in X\}$   $\triangleright$  set initialization
11:   $\triangleright$  get the candidate using Hopkins statistic
12:   $Y \leftarrow \{y : y \in X \wedge Y \subseteq X\}$   $\triangleright$  uniformly distributed set
13:  for CandidateTerm  $t \in X$  do
14:     $rc \leftarrow \{\forall x : x \in S \wedge x \neq t\}$ 
15:     $HS[rc] \leftarrow \text{calculateHopkinsStatistic}(rc, X, Y, WE)$ 
16:    if  $HS[rc] \geq HS_{max}$  then
17:       $RC_{HS} \leftarrow rc$   $\triangleright$  candidate with maximum HS
18:       $S \leftarrow \{S \setminus t\}$ 
19:    end if
20:  end for
21:   $\triangleright$  get the candidate using polygon area
22:  for CandidateTerm  $t \in X$  do
23:     $rc \leftarrow \{\forall x : x \in S' \wedge x \neq t\}$ 
24:     $PA[rc] \leftarrow \text{calculatePolygonArea}(rc, WE)$ 
25:    if  $PA[rc] \leq PA_{min}$  then
26:       $RC_{PA} \leftarrow rc$   $\triangleright$  candidate with minimum PA
27:       $S' \leftarrow \{S' \setminus t\}$ 
28:    end if
29:  end for
30:   $\triangleright$  get the best reformulation using machine learning
31:   $RC_{best} \leftarrow \text{getBestQR}(RC_{HS}, RC_{PA}, RC_{base})$ 
32:   $Q^R \leftarrow \{Q_{pp} \cup RC_{best}\}$ 
33:   $\triangleright$  bug localization on codebase with reformulated query  $Q^R$ 
34:   $R \leftarrow \text{Lucene}(\text{corpus}, Q^R)$ 
35:  return  $R$ 
36: end procedure
```

---

### 6.3.2 Automated Search Query Reformulation with Semantic Hyperspace, Clustering Tendency & Machine Learning

We use the semantic hyperspace constructed above (Section 6.3.1), identify the best reformulation candidate using two clustering tendency metrics and machine learning, and then expand a poor query intended for IR-based bug localization as follows:

**Selection of Candidate Source Terms:** First step of automatic query reformulation is to collect suitable candidate terms for the reformulation. Existing approaches from the literature often make use of relevance feedback [84, 146] or pseudo-relevance feedback [62, 98, 213, 222] for candidate selection. Rahman and Roy [189] recently make use of *field* and *method signatures* from the source code of a software system, and suggest suitable terms for query reformulation using pseudo-relevance feedback and an advanced term weighting method (e.g., PageRank [57]). Their approach also outperforms contemporary PRF-based approaches which makes it an ideal choice for our candidate term selection. We use authors’ implementation of the tool, and extract the Top-25 terms (i.e., *threshold* justified in RQ<sub>1</sub>) from the source code as the candidate terms (i.e., Step 4, Fig. 6.1, Lines 4–8, Algorithm 7). Given that low quality bug reports often lack relevant program entities, such terms from the relevant source code could complement them for bug localization.

**Clustering Tendency Analysis:** Once candidate terms are selected, we attempt to identify the most appropriate ones from them to reformulate a given query (i.e., bug report). Several earlier studies [194, 268, 274] simply rely on *semantic distance* between query keywords and candidate terms for choosing the appropriate terms for query reformulation. However, their idea might fail with poor search queries (i.e., low quality bug reports). While a term from the source code could be semantically close to one of the query keywords, (1) it might not be relevant to the whole query or (2) the keyword itself might not be a salient one. We thus leverage the *clustering tendency* of the candidate terms towards the query keywords rather than simply relying on their semantic distance. In particular, we locate the query keywords and the candidate terms within our *semantic hyperspace* using their embedding vectors (i.e., *coordinates*), determine their clustering tendency with each other using two metrics below, and then develop two reformulation candidates as follows (Steps 5–6, Fig. 6.1, Lines 9–29, Algorithm 7):

(a) **Hopkins Statistic** is a statistical hypothesis test that determines the clustering tendency of a given dataset [48, 108]. It assumes a *null hypothesis* that the data points in the dataset have a *uniform random distribution*, and thus do not form any cluster. We use this statistic to identify a subset of the candidate terms that have the highest clustering tendency with a given query.

We first combine query keywords and candidate source terms, and develop a sample set  $X$  of size  $n$ . For the sake of brevity, we assume that  $X$  contains  $n$  items. Now, we construct a subset  $S \subseteq X$  of size  $m \ll n$  by iteratively discarding the individual items. We also draw a *uniform random* sample  $Y \subseteq X$  of size  $l \ll n$  where  $Y$  contains only  $l$  distinct items (i.e., uniformly distributed). That is,  $S$  is the real dataset and  $Y$  is the uniformly sampled set. We then determine the distance between each item and its nearest neighbour

from  $X$ . Lets assume that  $u_i$  is the distance between  $x_i \in S$  and its nearest neighbour in  $X$  whereas  $v_i$  is the distance between  $y_i \in Y$  and its the nearest neighbour in  $X$ . Now, we calculate the Hopkins statistic  $HS$  for  $S$  using the distance measures as follows:

$$HS = \frac{\frac{1}{l} \sum_{i=1}^l v_i^d}{\frac{1}{m} \sum_{i=1}^m u_i^d + \frac{1}{l} \sum_{i=1}^l v_i^d} \quad (6.1)$$

Here,  $d$  refers to the dimension of each data point. Since we represent each item  $x_i \in X$  using corresponding embedding vector with a size of  $d$ , the distance measures above are calculated using *cosine similarity* [90]. Let us assume that two items (i.e., terms)  $r \in S$  and  $t \in X$  have two embedding vectors  $\mathbf{R}$  and  $\mathbf{T}$  respectively which are derived from our semantic hyperspace (i.e., Section 6.3.1). Now, the semantic distance  $u_i^d$  between  $r$  and  $t$  is calculated as follows:

$$u_i^d = 1 - \frac{\sum_{k=1}^d S_k \times T_k}{\sqrt{\sum_{k=1}^d S_k^2} \sqrt{\sum_{k=1}^d T_k^2}} \quad (6.2)$$

Hopkins statistic takes a value between 0 and 1. A value close to 1 indicates that the sample set  $S$  is highly clustered whereas 0.5 indicates that  $S$  is randomly sampled, and does not contain any meaningful clusters. We construct a number of subsets  $S \subseteq X$ , and choose the one with the highest  $HS$  value as our reformulation candidate  $RC_{HS}$  (i.e., Step 6a, Fig. 6.1, Lines 11–20, Algorithm 7).

**(b) Polygon Area** calculation, a well known concept from *Coordinate Geometry*, has found numerous applications in the real world problems (e.g., architectural planning, computer 3D modelling) [34]. These problems often involve the maximization or minimization of the area of an irregular polygon. In order to reformulate a poor query, we deal with a set of points within the semantic hyperspace which also essentially form an irregular polygon. Carmel et al. [61] suggest that poor queries often discuss multiple topics which make them ambiguous, and good quality queries are mostly precise. Similarly, we argue that terms of a good quality search query are closely related in their semantics. Hence, the area of polygon created by them within our semantic hyperspace is likely to be *small*. Conversely, a large polygon area indicates broad or ambiguous query topics. Thus, the polygon area could be another *proxy* to *clustering tendency* between the candidate reformulation terms and the query keywords (e.g., bug report).

We develop a sample set  $X$  by combining candidate terms and query keywords. For the sake of brevity, let us assume that  $X$  contains  $n$  points where each point is an embedding vector of size  $d$  for corresponding term or keyword from  $X$ . Since we deal with a  $d$ -dimensional space (a.k.a., hyperspace), we adapt the traditional 2-dimensional polygon area calculation [33] with  $d$ -dimensions. In particular, we choose  ${}^d C_2$  dimension pairs, calculate the polygon area for each pair, and then sum them up to obtain the final area  $PA$ . Let us assume

that  $R$  and  $T$  contain  $r^{th}$  and  $t^{th}$  coordinates from  $n$  points above. Now we calculate their polygon area using the following equation.

$$PA_{RT} = \frac{1}{2} \sum_{i=1}^n (R_j + R_i) \times (T_j - T_i) \quad \text{where } j = i - 1 \quad (6.3)$$

Here,  $j$  takes a value of  $n$  when  $i = 1$ , otherwise it always takes the previous value of  $i$ , i.e.,  $j = i - 1$ . We construct a number of subsets  $S \subseteq X$ , extract their embedding vectors, and calculate the polygon area for each subset. Then, we choose the one with the smallest polygon area as the reformulation candidate  $RC_{PA}$  (i.e., Step 6b, Fig. 6.1, Lines 21–29, Algorithm 7).

**Selection of the Best Reformulation Candidate:** The clustering tendency metrics above provide two reformulation candidates ( $RC_{HS}$ ,  $RC_{PA}$ ) using the candidate terms from the source code of a software system. An earlier study [193] suggests that baseline queries might also perform well in some cases. We thus use the preprocessed version of a given bug report as the third reformulation candidate  $RC_{base}$  (Step 6c, Fig. 6.1). Existing evidence [98, 189] shows that combination of multiple reformulation strategies performs consistently higher than any single strategy. We thus consider all three reformulation candidates, and choose the best one among them using machine learning (Lines 30–32, Algorithm 7) as follows:

**(a) Data Resampling:** Query difficulty metrics [62] have been used by several existing studies [96, 98, 189] to identify the best reformulation candidate. However, they might not be effective for poor queries (our problem context) since they mostly rely on the linguistic aspect of the queries. We thus attempt to predict the best reformulation candidate using the above two data analytics based metrics – *Hopkins Statistic*, and *Polygon Area* – of each candidate. We also determine *Query Effectiveness* of each of the three reformulation candidates, and annotate them with one of these three labels – ‘*high*’, ‘*medium*’, and ‘*low*’ – based on their performances. Since only ‘*high*’ labelled candidates are of our interest, the training dataset is inherently skewed. We thus perform *bootstrapping* (i.e., random resampling [116]) on the training dataset with 100% *sample size* and with *replacement* option. Bootstrapping is often used to mitigate data skewness [116]. In particular, we draw 50 random samples (i.e., suggested by an earlier study [189]) from this dataset, and prepare multiple datasets for our model training (Step 7, Fig. 6.1).

**(b) Machine Learning:** Given multiple training datasets, we develop multiple machine learning models, and then identify the best reformulation candidate by combining the predictions from each of these models. Such an approach of combining multiple models is known as *ensemble learning* [174, 214]. It is often used when individual models might be weak. As shown by earlier studies [56, 203], RandomForest algorithm has the potential to avoid model overfitting which makes it a suitable choice for our machine learning task. Besides, our investigation in RQ<sub>1</sub>, Section 6.4.3 shows that RandomForest is the best choice for our dataset. We thus train RandomForest model with 10-fold cross validations on each of the training datasets, combine predictions

from all the models for a given test instance, and then suggest the best candidate as the reformulated query (i.e., Steps 7–9, Fig. 6.1).

### 6.3.3 Bug Localization

Once BLADER returns a reformulated query against a poor search query (i.e., low quality bug report), we submit the reformulated query to a widely used code search engine namely *Lucene* [32]. Lucene employs Boolean Search Model and Vector Space Model [114] for the search, and it is widely adopted both by the industry (e.g., ElasticSearch) and by the existing literature [98, 164, 189]. We use Okapi BM25 similarity [227] between a query and the source code documents, and retrieve a ranked list of buggy source documents for each query using Lucene (i.e., Lines 33–35, Algorithm 7). These ranked buggy documents are then presented to the developer for manual analysis. As shown in Table 6.1, our suggested query returns the first buggy document at the **third** position as opposed to the 12<sup>th</sup> position by the given query.

## 6.4 Experiment

We evaluate our proposed technique –BLADER– with *four* widely used performance metrics and 1,546 bug reports from *six* subject systems. We further consider *two* different dimensions – (1) bug localization and (2) query reformulation– for our evaluation. We first determine our bug localization performance (i.e., BLADER<sub>BL</sub>), and compare with one baseline approach and five existing approaches on bug localization [192, 220, 250, 268, 276] including the state-of-the-art [192]. In addition to that, we contrast our query reformulation performance (i.e., BLADER<sub>QR</sub>) with that of another six existing approaches on query reformulation [188, 191, 192, 212, 213, 231] including the state-of-the-art [192]. Thus, we answer four research questions using our experiments as follows:

- **RQ<sub>1</sub>**: Can BLADER<sub>BL</sub> outperform the baseline approach in bug localization? Are the adopted thresholds, parameters and choices justified?
- **RQ<sub>2</sub>**: Can BLADER<sub>QR</sub> outperform the baseline approach in query reformulation intended for bug localization?
- **RQ<sub>3</sub>**: Can BLADER<sub>BL</sub> outperform the state-of-the-art in the IR-based bug localization?
- **RQ<sub>4</sub>**: Can BLADER<sub>QR</sub> outperform the state-of-the-art in query reformulation intended for bug/concept location?

### 6.4.1 Experimental Dataset

**Dataset Collection:** We use a total of 1,546 bug reports from six open source, Java-based systems for the evaluation and validation of our technique. We collect these bug reports from a publicly available *benchmark dataset* [30, 192, 193]. Table 6.2 shows the details of our dataset. First, all the resolved bug reports (i.e.,



marked as RESOLVED) are collected from either BugZilla or JIRA repository of each system. Second, bug-fixing commits (i.e., commits solving the bugs) are extracted from the version control history of each system at GitHub using a set of appropriate regular expressions [43]. Such bug reports are then selected that have the corresponding bug-fixing commits. Third, we retain such bug reports that contain only unstructured regular texts, and do not contain any structured entities (e.g., class names, stack traces), i.e.,  $BR_{NL}$  category, *poor bug reports* [192, 248]. Fourth, we also discard such bug reports for which (1) no source code documents (e.g., Java classes) are changed, and (2) the changed source code documents do not exist any more in the current snapshot of the subject system.

**Construction of Ground Truth:** We extract the *changeset* (i.e., list of changed files) from each of the bug-fixing commits, and use them as the *ground truth* for corresponding bug reports. When multiple commits are found for the same bug report, their changesets are merged together to form the ground truth. Recent studies [103, 260] report concerns about *tangled commits* that often contain changes irrelevant to the fixed bug. In order to mitigate such threat, we also discard the large commits that contain more than *five* changed documents from the dataset. It should be noted that evaluations are performed both with and without tangled commits.

**Replication Package:** Our dataset, experimental results and working prototype are *publicly available* (<https://goo.gl/tcVKup>) for the replication or third party reuse.

## 6.4.2 Performance Metrics

We use four state-of-the-art performance metrics that are frequently adopted for evaluation by the existing approaches on IR-based bug localization [163, 220, 250] and on query reformulation [98, 164, 189]. They are defined as follows:

**Hit@K / Top-K Accuracy** determines the fraction of all queries by each of which at least one buggy source document is retrieved within the Top-K positions of result list [191, 220, 250]. The higher the measure is, the better the approach (or its query) is.

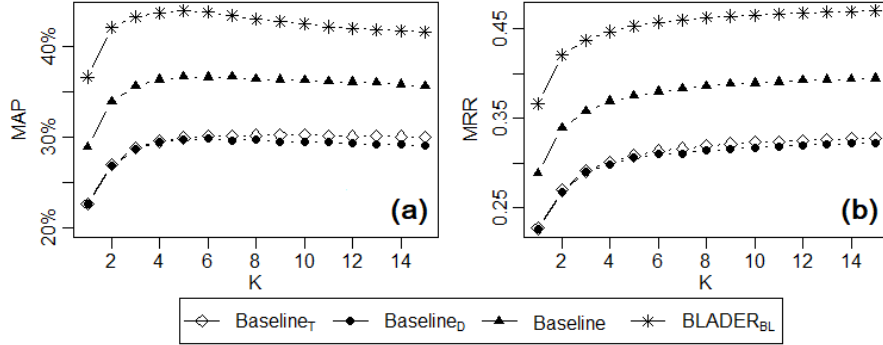
**Mean Average Precision@K (MAP)** considers the rank of the results unlike the traditional precision measure [164, 220, 276]. Precision@K calculates the precision of an approach at the occurrence of  $k^{th}$  result. Average Precision@K (AP@K) averages the Precision@K from all the true positive results (i.e., buggy source documents) within a result list. Mean Average Precision@K (MAP) averages AP@K over all the queries in the dataset ( $Q$ ) as follows:

$$AP@K = \frac{\sum_{k=1}^K P_k \times buggy(k)}{|S|}, \quad MAP(Q) = \frac{\sum_q AP@K(q)}{|Q|}$$

Here,  $P_k$  refers to Precision@K,  $K$  is the number of top results under analysis, and  $S$  is the set of true positive results retrieved by each query. The function  $buggy(k)$  returns a value of 1 when  $k^{th}$  result from the list is true positive (i.e., actually buggy) and 0 for the opposite. The higher the MAP measure is, the better the approach (or its query) is. We calculate MAP for top 1 to 15 results only.

**Table 6.2:** Experimental Dataset (Subject Systems & Bug Reports)

System	Duration	#BR	System	Duration	#BR
ecf	2001–2017	163	eclipse.jdt.ui	2001–2016	407
eclipse.jdt.core	2001–2016	132	eclipse.pde.ui	2001–2016	510
eclipse.jdt.debug	2001–2017	229	tomcat70	2001–2016	105
<b>Total: 1,546</b>					

**Figure 6.2:** Comparison of our approach, BLADER<sub>BL</sub>, with the baseline approach in bug localization using (a) MAP and (b) MRR

**Mean Reciprocal Rank (MRR)** is defined as the mean of Reciprocal Rank of all queries ( $Q$ ). Reciprocal Rank is the multiplicative inverse of the rank of the first buggy document within the results retrieved by each query [163, 220, 276]. For the sake of practicality, we only consider top 1 to 15 results retrieved by the query for our analysis. Thus, MRR can be calculated as follows:

$$MRR(Q) = \frac{1}{|Q|} \sum_{q \in Q} \frac{1}{firstRank(q)}$$

Here,  $firstRank(q)$  returns the rank of the first buggy document retrieved by a query  $q$ . The bigger the measure is, the better the approach (or its query) is.

**Query Effectiveness (QE)** is defined as the rank of the first buggy source document retrieved by a query [98, 163, 164]. It should be noted that all retrieved results are analysed in this case rather than Top-K only. The measure approximates a *developer’s effort* in locating the first buggy document in the result list. The higher the rank is, the quicker a developer locates the first buggy source document. Thus, a small QE value indicates high quality of a query.

### 6.4.3 Evaluation of BLADER

We first show the bug localization performance of our technique using the metrics above (Section 6.4.2), and then compare with a baseline approach and two of its variants. Like several earlier studies [98, 164, 192, 220], our baseline uses a pre-processed version of the bug report as a query, and employs *Lucene* [32] as the document retrieval engine. Thus, our **Baseline** = (pre-processed bug report texts + Lucene). We answer RQ<sub>1</sub> and RQ<sub>2</sub> as follows:

**Table 6.3:** Performance of BLADER<sub>BL</sub> in Bug Localization

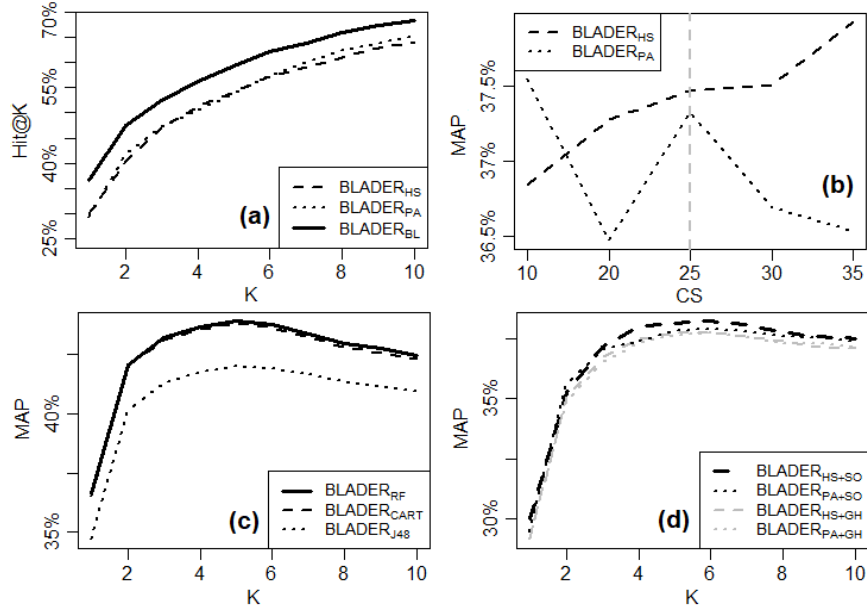
Technique	Hit@1	Hit@5	Hit@10	MAP	MRR
Baseline <sub>T</sub>	22.73%	45.22%	56.35%	30.30%	0.32
Baseline <sub>D</sub>	22.65%	44.55%	53.60%	29.49%	0.32
Baseline	28.89%	52.10%	62.54%	36.23%	0.39
<b>BLADER<sub>BL</sub></b>	<b>*36.64%</b>	<b>*59.39%</b>	<b>*68.29%</b>	<b>*42.46%</b>	<b>*0.47</b>
Performance of BLADER <sub>BL</sub> without Tangled Commits					
Baseline <sub>T</sub>	19.27%	42.69%	53.59%	27.87%	0.29
Baseline <sub>D</sub>	19.66%	41.44%	49.76%	27.66%	0.29
Baseline	25.90%	49.12%	60.32%	34.73%	0.36
<b>BLADER<sub>BL</sub></b>	<b>*32.97%</b>	<b>*55.95%</b>	<b>*65.56%</b>	<b>*41.11%</b>	<b>*0.43</b>

\* = Significantly higher than Baseline, **Baseline** uses *whole texts* (title + description), **Baseline<sub>T</sub>** uses only *title*, and **Baseline<sub>D</sub>** uses only *description* from a bug report

**Answering RQ<sub>1</sub>-(a) Performance of BLADER<sub>BL</sub> in Bug Localization:** Table 6.3 (upper part) shows Top-1, Top-5 and Top-10 performances of our technique. While the baseline achieves a maximum Hit@1 of 29%, our approach returns the correct results (i.e., buggy source documents) at the Top-1 position for 37% of the queries (i.e., 37% Hit@1) which is 27% higher than the baseline measure. However, the main strengths of our approach, BLADER<sub>BL</sub>, are in precision and reciprocal rank. Baseline approach achieves a precision of 36% and a reciprocal rank of 0.39 when Top-10 results are analysed. On the contrary, our approach achieves a precision of 42% and a reciprocal rank of 0.47 in the same case which are 17% and 21% higher respectively. Statistical tests such as *Wilcoxon Signed Rank* and *Cliff’s Delta* tests (across six subject systems) also report strong significance with *p-values* < 0.05 and a *large effect size* (i.e.,  $\delta=0.78$ ). Fig. 6.2 also clearly demonstrates that BLADER<sub>BL</sub> achieves higher precision and higher reciprocal rank than those of the baseline and its two variants for top 1 to 15 results.

Several earlier studies [103, 260] voice concerns about *tangled commits* that fix bugs but contain changes unrelated to the bugs. We discard such commits, and repeat our experiments. From Table 6.3 (lower part), we see that our approach performs significantly higher than the baseline even with this refined dataset. For example, BLADER<sub>BL</sub> achieves 18% higher precision and 19% higher reciprocal rank than those of the baseline. All these empirical findings above suggest that BLADER<sub>BL</sub> has higher potential than the baseline approach and its two variants for the IR-based bug localization.

**Answering RQ<sub>1</sub>-(b) Impact of Adopted Thresholds, Parameters and Choices:** We conduct several experiments to justify our adopted thresholds, parameters and choices – (a) use of multiple reformulation candidates, (b) number of candidate terms, (c) use of machine learning algorithm, and (d) use of Stack Overflow for learning word embeddings. From Fig. 6.3-(a), we see that reformulation candidates based on



**Figure 6.3:** Impact of our adopted thresholds, parameters and choices – (a) Multiple reformulation candidates, (b) Number of candidate source terms, (c) Machine learning algorithm for the best query selection, and (d) Corpus for learning word embeddings

Hopkins Statistic (i.e.,  $BLADER_{HS}$ ) and Polygon Area (i.e.,  $BLADER_{PA}$ ) perform similarly. However, our approach,  $BLADER_{BL}$ , leverages both these candidates, delivers the best candidate using machine learning (Steps 6–9, Fig. 6.1), and thus achieves comparatively higher Hit@K performance. Such finding justifies the use of multiple reformulation candidates. Similar evidence was found in the earlier studies [98, 189] as well. Fig. 6.3-(b) shows that our approach performs optimally when Top-25 terms are collected from the project source code as the candidate terms (Step 4b, Fig. 6.1). We also conduct experiments using three machine learning algorithms – *CART* [98], *J48* [180] and *RandomForest* [56]– for identifying the best reformulation candidate (Section 6.3.2, Step 8, Fig. 6.1). As shown in Fig. 6.3-(c), RandomForest based approach (i.e.,  $BLADER_{RF}$ ) performs better than  $BLADER_{J48}$  and marginally better than  $BLADER_{CART}$ . RandomForest is also more robust to model over fitting problem [56]. Thus, our choice of using RandomForest as the machine learning algorithm is possibly justified. We also investigate whether the use of corpus in learning the word embeddings matters or not (Section 6.3.1, Steps 1–3, Fig. 6.1). We conduct experiments using two open source corpora - Stack Overflow [194] and GitHub [90]. As shown in Fig. 6.3-(d), Stack Overflow based versions, (i.e.,  $BLADER_{HS+SO}$ ), perform marginally higher than the GitHub based versions. Such finding also justifies our choice of using Stack Overflow for learning the word embeddings.

**Summary of RQ<sub>1</sub>:** Our technique,  $BLADER_{BL}$ , achieves **27%** higher accuracy, **17%** higher precision and **21%** higher reciprocal rank than those of the baseline approach. Furthermore, our adopted thresholds, parameters and choices are justified.

**Table 6.4:** Comparison of Query Effectiveness with Baseline Queries

Query Pairs	Improved	Worsened	Preserved
<b>Comparison with All Queries (1,546)</b>			
BLADER <sub>QR</sub> vs. BQ	735 ( <b>47.54%</b> )	182 (11.77%)	629 ( <b>40.69%</b> )
<b>Comparison with Low Quality Baseline Queries (569)</b>			
BLADER <sub>QR</sub> vs. BQ	410 ( <b>72.06%</b> )	85 (14.94%)	74 (13.01%)
<b>Comparison without Tangled Commits (1,074)</b>			
BLADER <sub>QR</sub> vs. BQ	537 ( <b>50.00%</b> )	134 (12.48%)	403 ( <b>37.52%</b> )

**BQ = Baseline Queries**

**Answering RQ<sub>2</sub>–Comparison with the Baseline Queries:** Although RQ<sub>1</sub> shows high potential of our approach, we further compare with the baseline queries. In particular, we compare the rank of the first correct result (i.e., buggy source document) returned by a baseline query with such rank of the corresponding reformulated query from BLADER<sub>QR</sub>. If the rank of BLADER<sub>QR</sub> is higher than that of the baseline, we call it *query improvement*, and the opposite as *query worsening*. On the contrary, if both ranks are equal, we call it *query preserving*. From Table 6.4, we see that BLADER<sub>QR</sub> improves 48% of the baseline queries, keeps the quality of 41% queries unchanged, and worsens only 12% of the queries. That is, on average, nine out of ten baseline queries (i.e.,  $90\% \approx 48\% + 41\%$ ) either get improved or get preserved through the reformulations offered by BLADER<sub>QR</sub>, which are highly promising according to relevant literature [98, 191, 231]. Result rank distributions shown in Table 6.6 also suggest that the ranks achieved by BLADER<sub>QR</sub> are closer to the top of the list than those of the baseline. We also repeat our experiments using very low quality baseline queries which return their first correct results below the 10<sup>th</sup> position of the list (i.e., QE>10). As shown in Table 6.4, BLADER<sub>QR</sub> improves 72% of these low quality queries. Similar findings are also observed for the queries without tangled commits (e.g., 50% improvement). All these empirical findings above clearly demonstrate the high potential of our reformulated queries over the baseline queries.

**Summary of RQ<sub>2</sub>:** About **48%–72%** of the queries provided by our technique perform better than the corresponding baseline queries (taken from bug reports). Nine out of ten baseline queries either get improved or get preserved due to the reformulations of BLADER<sub>QR</sub>.

#### 6.4.4 Comparison with Existing Techniques

Although our approach demonstrates high potential over the baseline approach both in RQ<sub>1</sub> and RQ<sub>2</sub>, we further compare with the state-of-the-art in order to place our work in the literature. In particular, we extensively compare with five existing studies on IR-based bug localization [192, 220, 250, 268, 276] including the state-of-the-art [192] and six existing studies on query reformulation [188, 191, 192, 212, 213, 231]. We answer RQ<sub>3</sub> and RQ<sub>4</sub> as follows:

**Table 6.5:** Comparison with Existing Bug Localization Techniques

Technique	Hit@1	Hit@5	Hit@10	MAP	MRR
<b>Basic and Structured Information Retrieval</b>					
BugLocator	25.04%	48.51%	58.90%	32.11%	0.35
BLIZZARD	29.16%	53.78%	65.21%	37.62%	0.40
BLUiR	29.91%	56.59%	66.10%	38.11%	0.41
BLADER <sub>BL</sub>	<b>*36.64%</b>	<b>59.39%</b>	<b>68.29%</b>	<b>*42.46%</b>	<b>*0.47</b>
<b>Information Retrieval + Multiple Ranking Algorithms</b>					
AmaLgam <sub>BRO</sub>	29.40%	56.07%	65.01%	37.74%	0.40
BLIZZARD <sub>BRO</sub>	35.45%	58.75%	69.17%	42.26%	0.46
BLADER <sub>BRO+BL</sub>	35.44%	<b>61.27%</b>	<b>70.63%</b>	<b>42.97%</b>	<b>0.47</b>
<b>Information Retrieval + Multiple Ranking + External Resources</b>					
AmaLgam+	49.72%	65.42%	71.49%	52.74%	0.57
BLIZZARD+	47.97%	66.24%	74.49%	52.12%	0.56
BLADER <sub>+</sub> BL	48.39%	<b>67.60%</b>	<b>75.70%</b>	52.62%	<b>0.57</b>
<b>Information Retrieval + Word Embeddings</b>					
Ye et al.	21.74%	43.36%	55.33%	29.24%	0.31
BLADER <sub>BL</sub>	<b>*36.64%</b>	<b>*59.39%</b>	<b>*68.29%</b>	<b>*42.46%</b>	<b>*0.47</b>

\*=Significantly higher than baseline, **Emboldened**=Comparatively higher

**Answering RQ<sub>3</sub> – Comparison with IR-Based Bug Localization Techniques:** Zhou et al. [276] first use an improved version of Vector Space Model (a.k.a., rVSM), and localize software bugs with Information Retrieval (IR). Saha et al. [220] introduce structured Information Retrieval where they leverage the structures of both bug reports and source code documents for the bug localization. Later studies [249, 258] combine both rVSM and structures. In the same vein, Wang and Lo [250] combine five items from the literature – similar bug reports, structured IR, stack traces, version control history and author history, and outperform earlier approaches. However, recently, our another work [192] introduces novel aspects such as *report quality dynamics* and query reformulation, and outperforms earlier approaches in the IR-based bug localization which makes it *state-of-the-art*. Ye et al. [268] recently combine semantic similarity and textual similarity for IR-based bug localization. These five works above namely BugLocator [276], BLUiR [220], AmaLgam+ [250], BLIZZARD [192] and Ye et al. [268] are highly relevant to ours, and we compare with them experimentally. We collect the authors’ implementations of three techniques [192, 220, 276], and carefully re-implement the remaining two [250, 268] by consulting with their original authors for the comparison.

From Table 6.5 (upper part), we see that two of the existing approaches –BLUiR and BLIZZARD– that rely on structured Information Retrieval stand out from the rest. BLUiR achieves a maximum precision of 38% and a maximum reciprocal rank of 0.41 when Top-10 results are analysed. On the contrary, our approach, BLADER<sub>BL</sub>, achieves 11% higher precision and 15% higher reciprocal rank than BLUiR in the same context. Fig. 6.4 further contrasts the precision and reciprocal rank of BLADER<sub>BL</sub> with that of the

**Table 6.6:** Comparison of Query Effectiveness with Existing Query Reformulation Techniques

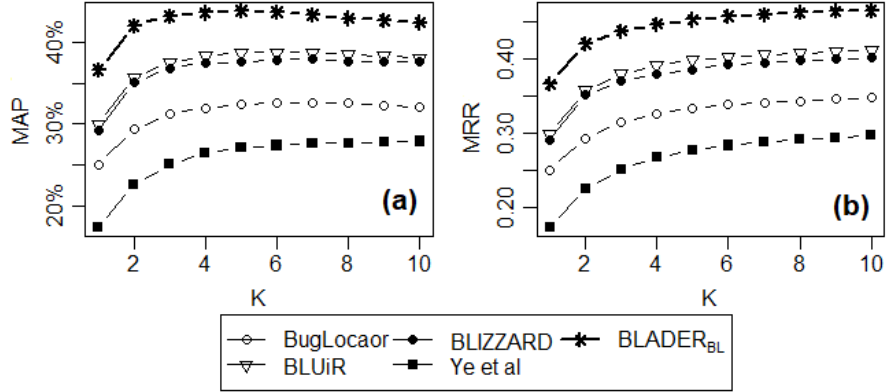
Technique	Improvement						Worsening						Preserving	
	#Improved	Mean	Q1	Q2	Q3	Max.	#Worsened	Mean	Q1	Q2	Q3	Min.	Max.	#Preserved
Rocchio [213]	32 (2.07%)	33	4	8	19	365	24 (1.55%)	140	4	12	146	2	850	1,490 (96.38%)
RSV [212]	345 (22.27%)	112	3	9	38	6,564	751 (48.57%)	105	7	23	81	2	2,140	450 (29.11%)
QUICKAR [188]	395 (25.55%)	65	2	7	31	2,773	835 (54.01%)	125	6	24	101	2	2,761	313 (20.25%)
Sisman and Kak [231]	499 (32.28%)	59	2	6	25	2,019	575 (37.19%)	98	5	16	64	2	2,204	472 (30.53%)
STRICT [191]	467 (30.21%)	57	2	6	30	1,213	654 (42.30%)	112	5	18	63	2	4,933	425 (27.49%)
BLIZZARD [192]	<b>597 (38.62%)</b>	75	2	8	32	3,063	455 (29.43%)	92	5	15	54	2	2,024	494 (31.95%)
Baseline	-	84	5	13	49	2,434	-	79	2	9	47	1	1,894	-
<b>BLADER<sub>QR</sub></b>	<b>735 (47.54%)</b>	<b>56</b>	<b>2</b>	<b>7</b>	<b>24</b>	<b>2,509</b>	<b>182 (11.77%)</b>	<b>125</b>	<b>5</b>	<b>16</b>	<b>82</b>	<b>2</b>	<b>2,444</b>	<b>629 (40.69%)</b>

Mean = Mean rank of first correct results returned by the queries,  $Q_i = i^{th}$  quartile of all ranks considered

**Table 6.7:** Comparison with Existing Studies using Feature Matrix

Technique	Report Content		QR	Data Analytics		SVA	MRR
	RT	RS		SS	CTA		
Baseline	●						0.39
BugLocator	●						0.35
BLUiR	●	●					0.41
BLIZZARD	●		●				0.40
Ye et al.	●			●			0.31
BLADER <sub>BL</sub>	●		●		●		0.47
AmaLgam+	●	●				●	0.57
BLIZZARD+	●	●	●			●	0.56
BLADER <sub>BL</sub> +	●	●	●		●	●	0.57

RT=Report Texts, RS=Report Structure, QR=Query Reformulation, SS=Semantic Similarity, CTA=Clustering Tendency Analysis, SVA=Similar reports, Version control history, and Author history, ●=Feature used

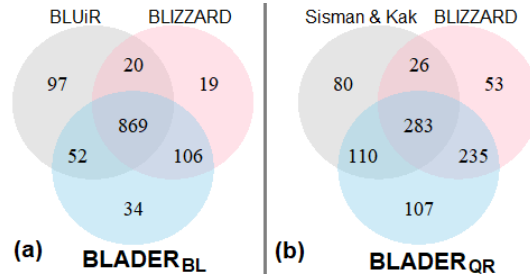


**Figure 6.4:** Comparison of our approach with the existing techniques in bug localization using (a) MAP, and (b) MRR for top 1 to 10 results

state-of-the-art approaches for top 1 to 10 results. We see that BLADER<sub>BL</sub> outperforms four ( $m = 4$ ) existing approaches including the state-of-the-art with statistical significance. We perform *Wilcoxon Signed Rank* and *Cliff's delta* tests with *Bonferroni Correction*, and report each  $p\text{-value} \leq 0.0125$  ( $\alpha/m$ ) and a *large effect size* with  $0.84 \leq \delta \leq 1.00$ . Fig. 6.5-(a) shows the overlap of successfully localized bugs by BLADER<sub>BL</sub> and by the state-of-the-art approaches [192, 220]. We see that our approach localizes about 92% of the bugs that were identified by BLUiR and BLIZZARD, and then it localizes an additional 34 unique bugs when only Top-10 results are analysed. All these findings above suggest the high potential of our approach over the state-of-the-art especially in terms of precision and reciprocal rank.

From Table 6.5 (middle part), we also see that AmaLgam+, BLIZZARD+ and their variants achieve higher performances than that of BLADER<sub>BL</sub> by combining multiple ranking algorithms and by employing additional resources (e.g., past bug reports, version control history, author history). Although such perfor-



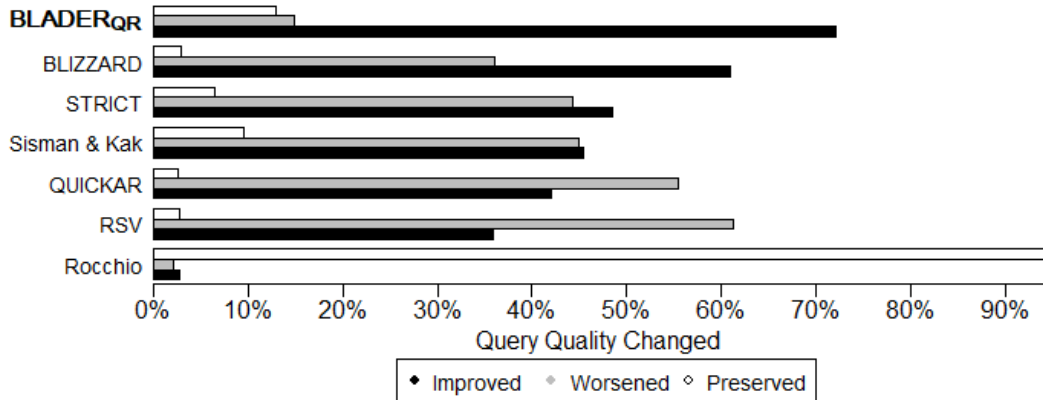


**Figure 6.5:** (a) Overlap of the successfully localized bugs between BLADDER<sub>BL</sub> and the state-of-the-art, and (b) Overlap of the improved queries between BLADDER<sub>QR</sub> and the state-of-the-art approaches

mance improvements are highly desirable, we argue that these approaches [249, 250] are not only less scalable but also limited in their usability. Majority of their performances come from third party items which could be unavailable. On the contrary, our goal is to *improve* the *basic IR-based localization* using the primary resources available at hand (e.g., bug reports and source code documents). Our localization is cheap, easy-to-use, and scalable. Nonetheless, when these third party items are considered, one of the variants of our approach, BLADDER<sub>+BL</sub>, performs equally or higher than the competing approaches [192, 250] (Table 6.5). We also compare with Ye et al. [268] that combines textual similarity and semantic similarity between a bug report and the source code document. Our investigation in Table 6.5 (lower part) demonstrates that simple semantic similarity might not be sufficient enough if the poor queries are not improved at the first place. Our approach BLADDER<sub>BL</sub> employs word semantics and clustering tendency analysis, complements the poor queries with appropriate keywords from the source code, and thus, outperforms Ye et al. in bug localization. Table 6.7 shows a matrix of eight different features that are used by the contemporary approaches. We add *clustering tendency analysis* (CTA) as a novel dimension there, overcome the challenges with poor search queries and outperform the state-of-the-art approaches [192, 220].

**Summary of RQ<sub>3</sub>:** Our approach outperforms five existing approaches on IR-based bug localization by a significant margin. BLADDER<sub>BL</sub> achieves **11%** higher precision and **15%** higher reciprocal rank than that of the state-of-the-art.

**Answering RQ<sub>4</sub> – Comparison with Query Reformulation Techniques:** Although our approach outperforms the state-of-the-art on bug localization, we further compare with six existing studies on query reformulation [188, 191, 192, 212, 213, 231]. Rocchio [213] and RSV [212] are two widely used query reformulation techniques in source code search [98, 251, 274]. STRICT [191] and QUICKAR [188] reformulate search queries for concept location by employing term weighting methods (e.g., TextRank) and crowdsourced knowledge from Stack Overflow respectively. Sisman and Kak [231] and BLIZZARD [192] reformulate queries for bug localization where they leverage *spatial code proximity* and *context-awareness* respectively. We use the authors’ implementation for QUICKAR, STRICT and BLIZZARD, carefully re-implement the remaining three [212, 213, 231] (i.e., unavailable prototypes), and then compare with their best performances on our dataset. We determine the rank of the first buggy document retrieved by each of the queries from each of



**Figure 6.6:** Comparison of our approach with the existing techniques in query reformulation using very low quality queries

these techniques, and then compare with corresponding baseline rank. If the rank of the reformulated query is higher than the baseline rank, we call it *query improvement* and the opposite as *query worsening*. From Table 6.6, we see that BLIZZARD, the state-of-the-art, improves 39% and worsens 29% of 1,546 baseline queries. On the contrary, our approach improves 48% and worsens 12% of the queries which are 23% higher and 60% lower respectively. Quartile analysis of the result ranks also demonstrates that our ranks are more promising (i.e., closer to the top) than the state-of-the-art ranks. Fig. 6.5-(b) shows the overlap of our improved queries with that of BLIZZARD and Sisman and Kak [231]. We see that our approach improves 78%–87% of the queries that were also improved by these two approaches. However, additional 107 unique queries are improved by BLADER<sub>QR</sub> where the earlier approaches totally fail. We further conduct experiments using 569 very poor queries which return their buggy source documents below the 10<sup>th</sup> position. As shown in Fig. 6.6, our approach improves the maximum amount of these poor queries (e.g., 72%) among all the approaches. Such ratios are 61% and 45% for BLIZZARD and Sisman and Kak respectively. Furthermore, BLADER<sub>QR</sub> worsens fewer queries (e.g., 12%–15%) than the existing approaches.

**Summary RQ<sub>4</sub>:** Our approach outperforms six existing approaches on query reformulation intended for concept/bug localization. BLADER<sub>QR</sub> improves **23%** more of the poor queries (i.e., low quality bug reports) than that of the state-of-the-art.

## 6.5 Threats to Validity

Threats to *internal validity* relate to experimental errors and biases [272]. Re-implementation of the existing approaches is a possible source of such threats. We re-implement five approaches [212, 213, 231, 250, 268] and reuse authors’ implementation of the remaining five approaches [188, 191, 192, 220, 276]. While we cannot rule out the possibility of human errors, we also partially validate our implementations against the previously published results [192, 250].

Threats to *external validity* relate to the generalizability of an approach [272]. Although we deal with Java-based systems and Java related Q & A threads from Stack Overflow, our approach can be easily replicated for other platforms. Our approach is not restricted by any programming language-specific features let it be semantic hyperspace construction or bug localization.

The use of data re-sampling during machine learning (Step 7, Fig. 6.1) might pose a threat to the reproducibility of our results. However, we employ *random seeds* to address this, repeat our experiments at least 50 times, and then use the average results. Thus, such a threat about reproducibility might also be mitigated. Furthermore, we have made our replication package publicly available [29]. Our query difficulty model might also be slightly biased towards the *high*-class candidates. Future work should adopt more rigorous methods for dealing with imbalanced data and focus on accurately predicting the best one from the list of reformulation candidates produced by our approach.

## 6.6 Related Work

**Bug Localization:** Automated bug localization has been one of the major challenges in Software Engineering over the last few decades [220, 248]. There have been a number of studies on bug localization which can be classified into two broad categories – *spectra based* and *Information Retrieval (IR) based* [248]. While spectra based approaches analyse execution traces, IR-based approaches leverage the textual similarity between bug report texts and source code documents for the bug localization. In this work, we deal with IR-based bug localization only. It is cheap, easy to use, and still provides good results [207, 248]. To date, existing studies make use of several Information Retrieval methods such as Latent Semantic Analysis (LSA) [150, 179], Latent Dirichlet Allocation (LDA) [167, 207] and Vector Space Model (VSM) [163, 192, 249, 250, 258, 276].

Zhou et al. [276] first introduce an improved version of Vector Space Model (VSM) namely rVSM, and localize software bugs using Information Retrieval (IR). Saha et al. [220] employ structured IR where they leverage the structures of both bug reports and source code documents for the localization task. Later studies [221, 249, 258] combine both rVSM and structured IR, and report improved performances. Wen et al. [255] also combine code level changes with basic IR for improved localization. Other studies combine spectral analysis [130, 142] and deep learning [128] with the basic IR. In the same vein, Wang and Lo [250] combine five items – similar bug reports [276], structured IR [220], stack traces [258], version control history [249], and author history, and outperform five earlier IR-based approaches. Rahman and Roy [192] recently integrate report quality dynamics and query reformulation, and their approach outperforms seven earlier approaches which makes it the *state-of-the-art* on IR-based bug localization. While most of these approaches combine additional ranking methods or external resources (e.g., version control history) with basic IR, we focus on improving the basic IR approach by reformulating the search query. From this point of view, Rahman and Roy [192] shares a similar goal like us. Unfortunately, their approach falls short in dealing with the poor queries as demonstrated by our and their experiments. We compare with five of the earlier studies on IR-based

bug localization [192, 220, 250, 268, 276] including the state-of-the-art [192], and the detailed comparison results can be found in Tables 6.5, 6.7, Fig. 6.4, and Section 6.4.4.

**Query Reformulation:** Existing studies on query reformulation employ term weighting [189, 191, 213], heuristics [64, 65, 104, 120, 231], pseudo-relevance feedback [213, 231], and machine learning [98, 189]. Rocchio [213] and RSV [212] are frequently used for query reformulation in Software Engineering contexts [98, 274]. STRICT [191] employs graph-based term weighting methods (e.g., TextRank, POSRank) to select appropriate search keywords from a bug report for concept location. QUICKAR [188] makes use of crowdsourced knowledge from Stack Overflow in the query reformulation task. Sisman and Kak [231] first introduce query reformulation in IR-based bug localization where they employ spatial code proximity (SCP) analysis. Recently, Rahman and Roy [192] integrate report quality dynamics into query reformulation, and their approach outperforms four earlier approaches which makes it the *state-of-the-art*. We compare with six of these existing approaches [188, 191, 192, 212, 213, 231] including the state-of-the-art [192], and the detailed comparison can be found in Table 6.6, Fig. 6.6, and Section 6.4.4. While the keyword selection method of Rahman and Roy works well with rich bug reports containing structured entities (e.g., stack traces), it generally fails with poor queries (i.e., low quality bug reports). On the contrary, our approach overcomes such challenge using word semantics, clustering tendency analysis and machine learning.

**Word Semantics:** Recent studies use word embeddings [54] in bug localization [128] and in code search [90, 194, 274] where they learn the embeddings from one or more large corpora (e.g., GitHub). However, all these studies simply rely on *semantic similarity* between a search query and the source code which might not be sufficient enough for dealing with the poor queries. According to our investigation (Table 6.5), poor queries need to be improved first before employing them for bug localization. Our approach exactly does that using appropriate keywords from the relevant source code, which are carefully chosen with word semantics (from Stack Overflow) and machine learning.

In short, while the existing approaches including the state-of-the-art fail to deal with poor queries (low quality bug reports), we overcome this crucial issue using word semantics, clustering tendency analysis, query reformulation and Information Retrieval. Such a comprehensive solution was not provided by any of the earlier studies from the literature.

## 6.7 Summary

Software bugs and failures cost billions of dollars every year [1]. Thus, resolution of the bugs or errors is a major part of software maintenance. Bug localization is a crucial step of the whole bug resolution process. Traditional solutions for bug localization do not perform well with poor quality bug reports (queries) since they do not contain any localization hints (e.g., program entities). We propose BLADER that overcomes this issue, and reformulates a given search query for bug localization using the underlying semantics of the keywords. In particular, our approach accepts a poor bug report as a search query, analyses the clustering

tendency between the query and the reformulation candidates in terms of their underlying semantics, and then delivers an improved, reformulated search query for the bug localization. We evaluate our technique using four performance metrics, two evaluation dimensions and 1,546 low quality bug reports (queries) from six subject systems. Our approach outperforms the state-of-the-art approach on IR-based bug localization by 11% higher precision and 15% higher reciprocal rank. Furthermore, our approach outperforms six existing approaches on query reformulation, and improves 23% more queries than the state-of-the-art.

During bug localization and concept location, software developers generally perform code search within a single software system. However, developers also frequently search for relevant code on the web, and reuse the retrieved code in various software maintenance tasks (e.g., new feature implementation). Thus, in essence, they perform code search within thousands of online software systems during the code search on the web (a.k.a., Internet-scale code search). Since each of our proposed approaches including this one (Chapters 3, 4, 5, 6) targets either concept location or bug localization, they might not be sufficient enough for query construction intended for Internet-scale code search. Online code repositories (e.g., GitHub, SourceForge) are much larger and noisier, and thus pose new challenges. In the next chapter, our fifth study (RACK, Chapter 7) overcomes these challenges, and reformulates a generic query for Internet-scale code search using the crowd generated knowledge from Stack Overflow Q&A site.

## CHAPTER 7

# SEARCH QUERY REFORMULATION FOR INTERNET-SCALE CODE SEARCH USING CROWDSOURCED KNOWLEDGE

Software maintenance costs about 60% of the total development time and efforts [88]. Bug localization and concept location are two major challenges of the maintenance phase. During bug localization and concept location, developers generally search for code within a single software system. However, developers also frequently search for relevant code on the web (a.k.a., Internet-scale code search) [55], and reuse the retrieved code in various software maintenance tasks (e.g., new feature addition). In this case, they search for relevant code within thousands of online software systems. Since each of our previous approaches (Chapters 3, 4, 5, 6) targets either concept location or bug localization (i.e., designed for single software system), they might not be sufficient enough for query construction intended for Internet-scale code search. Internet-scale code repositories (e.g., GitHub, SourceForge) are much larger and noisier, and thus pose new challenges in query construction and code search. In this chapter, we overcome this issue with another study. Here, we present RACK that accepts a programming task description as a query, reformulates the query with relevant API classes from Stack Overflow, and then delivers an improved, reformulated query for Internet-scale code search.

The rest of the chapter is organized as follows: Section 7.1 presents an overview of our study, Section 7.2 discusses the design and findings of our exploratory study, and Section 7.3 describes our proposed technique for search query reformulation. Section 7.4 discusses our evaluation and validation, and Section 7.5 identifies the threats to the validity of our findings. Section 7.6 discusses related work from the literature, and finally Section 7.7 concludes the chapter with future work.

### 7.1 Introduction

Studies show that software developers on average spend about 19% of their development time in web search [55]. On the web, they frequently look for relevant code snippets for their tasks [55, 263]. Online code search engines (e.g., Open Hub, Koders, GitHub) index thousands of large open source projects, and these projects are a potential source for such code snippets [44, 152]. However, these traditional, Internet-scale code search engines mostly employ keyword matching. Hence, they often do not perform well with unstructured natural language (NL) queries due to vocabulary mismatch between NL query and source code [45]. They retrieve code snippets based on lexical similarity between a search query and the project source code. That means,

these engines require the queries to be carefully designed by the users and to contain solution code-like information (e.g., relevant API classes). Unfortunately, preparing an effective search query that contains information on relevant APIs is not only challenging but also time-consuming for the developers [55, 120]. Previous studies [120, 125] also suggested that on average, developers regardless of their experience levels performed poorly in coming up with good queries for the code search. Thus, an automated technique that complements a natural language query with a list of relevant API classes or methods (i.e., search-engine friendly query) can greatly assist the developers in performing the code search. Our work addresses this particular research problem—*query reformulation with relevant API classes*—by exploiting the crowdsourced knowledge stored at Stack Overflow programming Q & A site.

Existing studies on API recommendation accept one or more natural language queries, and return relevant API classes and methods by mining feature request history and API documentations [243], large code repositories [274], API invocation graphs [63], library usage patterns [242], code surfing behaviour of the developers and API invocation chains [152]. McMillan et al. [152] first propose *Portfolio* that recommends relevant API methods for a given code search query and demonstrates their usage from a large codebase. Chan et al. [63] improve upon *Portfolio* by employing further sophisticated graph-mining and textual similarity techniques. Thung et al. [243] recommend relevant API methods to assist the implementation of an incoming feature request. Although all these techniques perform well in different working contexts, they share a set of limitations and thus fail to address the research problem of our interest. First, each of these techniques [63, 152, 243] exploits lexical similarity measure (e.g., Dice’s coefficients [63]) for candidate API selection. This warrants that the search query should be carefully prepared, and it should contain keywords similar to the API names. In other words, the developer should or must possess a certain level of experience with the target APIs to actually use these techniques [45]. Second, API names and search queries are generally provided by different developers who may use different vocabularies to convey the same concept [121]. Furnas et al. [83] named this the *vocabulary mismatch problem*. Lexical similarity based techniques often suffer from this problem. Hence, the performance of these techniques is not only limited but also subject to the identifier naming practices adopted in the codebase under study. We thus need a technique that overcomes the above limitations, and recommends relevant or appropriate APIs for natural language queries from a wider vocabulary.

One possible way to tackle the above challenges is to leverage the knowledge or experience of a large technical crowd on the usage of particular API classes and methods. Let us consider a natural language query—“*Generating MD5 hash of a Java string.*” Now, hundreds of Q&A threads from Stack Overflow discuss potential solutions and relevant APIs for this task which could be leveraged for our research. For instance, the Q&A example in Fig. 7.1 discusses on how to generate an MD5 hash (Fig. 7.1-(a)), and the accepted answer (Fig. 7.1-(b)) suggests that `MessageDigest` API should be used for the task. Such a usage of the API has also been recommended by at least 305 technical users from Stack Overflow, which validates the appropriateness of the usage. We leverage this crowd generated knowledge in relevant API suggestion. Our approach is thus generic, language independent, project insensitive, and at the same time, it overcomes the

**How can I generate an MD5 hash? (a)**


▲ Is there any method to generate MD5 hash of a string in Java?

491 `java` `hash` `md5` `hashcode`

▼ share edit flag

★ 130

edited Aug 13 '14 at 3:05  bjb568 5,078 ● 9 ● 22 ● 48

asked Jan 6 '09 at 9:45  Akshay 2,904 ● 4 ● 19 ● 26

---

20 Keep in mind that according to the recent research "MD5 should be considered cryptographically broken and unsuitable for further use". [en.wikipedia.org/wiki/MD5](http://en.wikipedia.org/wiki/MD5) – Zakharia Stanley May 3 '13 at 1:05

4 MD5 might be unsafe as a one-way security feature, but it is still good for generic checksum applications. – rustyx Feb 6 at 15:57

**27 Answers**

---

▲ `MessageDigest` is your friend. Call `getInstance("MD5")` to get an MD5 message digest you can use.

305 share edit flag

edited Aug 29 '11 at 19:48

answered Jan 6 '09 at 9:47  Bombe 38.9k ● 10 ● 82 ● 96

▼

✓

**(b)**

**Figure 7.1:** An example of (a) Stack Overflow question and (b) its accepted answer

vocabulary mismatch problem suffered by the past studies. One can argue in favour of Google which is often used by the developers for searching code on the web. Unfortunately, recent study [205] shows that developers need to spend more efforts (i.e., two times) in code search than in web search while using Google search engine. In particular, they need to reformulate their queries more frequently and more extensively for the code search. Such finding suggests that the general-purpose web search engines (e.g., Google) might be calibrated for the web pages only, and they perform sub-optimally with the source code, especially due to vocabulary mismatch issues [95, 102]. Thus, automatic tool supports in the query formulation for code search is still an open research problem that warrants further investigation.

In this chapter, we propose a novel query reformulation technique—RACK—that exploits the associations between query keywords and different API classes used in Stack Overflow and translates a natural language query intended for Internet-scale code search into a set of relevant API classes. First, we motivate our idea of using crowdsourced knowledge for API recommendation with an exploratory study where we analyse 172,043 questions and their accepted answers from Stack Overflow. Second, we construct a keyword-API mapping database using these questions and answers where the keywords (i.e., programming requirements) are extracted from questions and the APIs (i.e., programming solutions) are collected from the corresponding accepted answers. Third, we propose an API recommendation technique that employs three heuristics on keyword-API associations and recommends a ranked list of API classes for a given query for its reformulation.



The baseline idea is to capture and learn the responses from millions of technical users (e.g., developers, researchers, programming hobbyists) for different programming problems, and then exploit them for relevant API suggestion. Our technique (1) does not rely on the lexical similarity between a query and the API classes (or their documentations) for API selection, and (2) addresses the vocabulary mismatch problems by using a large vocabulary (i.e., 20K) produced by millions of users of Stack Overflow. Thus, it has a great potential for overcoming the challenges faced with the past studies.

An exploratory study with 172,043 Java related Q & A threads (i.e., question + accepted answer) from Stack Overflow shows that (1) each answer uses at least two different API classes on average (RQ<sub>1</sub>), and (2) about 65% of the classes from each of the 11 core Java API packages are used in these answers (RQ<sub>2</sub>). Such findings clearly suggest the potential of using Stack Overflow for relevant API suggestion. Experiments using 175 code search queries randomly chosen from three Java tutorial sites—*KodeJava*, *Java2s* and *Javadb*—show that our technique can recommend relevant API classes with an accuracy of 83%, a mean average precision@10 of 46% and a recall@10 of 54%, which are 66%, 79% and 87% higher respectively than that of the state-of-the-art [243] (RQ<sub>4</sub>, RQ<sub>8</sub>). Query reformulations using our suggested API classes improve 46%–64% of the baseline queries (i.e., contain natural language only), and their overall code retrieval accuracy improves by 19% (RQ<sub>9</sub>). Comparisons with the state-of-the-art techniques on query reformulation [168, 274] also demonstrate that RACK offers 48% net improvement in the baseline query quality as opposed to 26% by the state-of-the-art, which is 87% higher (RQ<sub>10</sub>). Our investigations with Google, Stack Overflow native search, and GitHub native code search also report that our reformulated queries can improve their results by 22%–26% in precision and 12%–28% in reciprocal rank in the context of code example search (RQ<sub>11</sub>).

**Novelty in Contribution:** This work is a significantly extended version of our earlier work [201] which employed two heuristics (KAC and KKC, Section III-B), experimented with 150 queries, and answered seven research questions. This work extends the earlier work in various aspects. First, we improve the earlier heuristics by recalibrating their weights and thresholds (i.e., RQ<sub>7</sub>). Second, we introduce a novel heuristic—Keyword Pair API Co-occurrence (KPAC, Section III-B)—that leverages word co-occurrences for candidate API selection more effectively. In fact, this one performs better than the earlier two. Third, we conduct experiments with a larger dataset containing 175 distinct queries, and further evaluate them in terms of their code retrieval performance (i.e., missing in the earlier work). Fourth, we extend our earlier analysis and answer 11 research questions (i.e., as opposed to seven questions answered by the earlier work). Fifth, we investigate the potential application of our approach in the context of code search using popular web search engines (e.g., Google) and Internet-scale code search engines (e.g., GitHub).

Thus, our work in this chapter makes the following contributions:

- (a) An exploratory study that suggests the potential of using Stack Overflow for relevant API suggestion against an NL query intended for Internet-scale code search.
- (b) A keyword-API mapping database that maps 655K natural language question keywords to 551K API classes from Stack Overflow Q&A site.

**Table 7.1:** API Packages for Exploratory Study

Package	#Class	Package	#Class
<b>Core Packages</b>			
java.lang	255	java.net	84
java.util	470	java.security	148
java.io	105	java.awt	423
java.math	09	java.sql	29
java.nio	189	javax.swing	1,195
java.applet	05		
<b>Non-Core Packages</b>			
java.beans	62	java.rmi	67
javax.xml	327	javax.annotation	17
java.text	44	javax.print	123
javax.sound	56	javax.management	201
<b>Total API Classes: 3,809</b>			

- (c) A novel query reformulation technique—RACK—that exploits query keyword-API associations stored in the crowdsourced knowledge of Stack Overflow, and reformulates a natural language query with a set of relevant API classes for Internet-scale code search.
- (d) Comprehensive evaluation of the proposed technique with six performance metrics, and comparison with the state-of-the-art techniques and contemporary web search engines (e.g., Google, Stack Overflow native search) and Internet-scale code search engines (e.g., GitHub native code search).

## 7.2 Exploratory Study

Our technique relies on the mapping between natural language keywords from the questions of Stack Overflow and API classes from corresponding accepted answers for translating a code search query into relevant API classes. Thus, an investigation is warranted whether such answers contain any API related information and the questions contain any search query keywords. We perform an exploratory study using 172,043 Q & A threads from Stack Overflow, and analyse the usage and coverage of standard Java API classes in them. We also explore if the question titles are a potential source of suitable keywords for code search. We particularly answer three research questions as shown in Table 7.2.

### 7.2.1 Data Collection

We collect 172,043 questions and their accepted answers from Stack Overflow using StackExchange data explorer [23] for our investigation. Since we are interested in Java APIs, we only collect such questions that

**Table 7.2:** Research Questions Answered using Exploratory Study

---

<b>Research Questions Targeting API Coverage</b>
<b>RQ<sub>1</sub>:</b> To what extent do the accepted answers from Stack Overflow refer to standard Java API classes?
<b>RQ<sub>2</sub>:</b> To what extent are the API classes from each of the core Java packages covered (i.e., mentioned) in the accepted answers from Stack Overflow?
<b>Research Question on Search Keyword Matching</b>
<b>RQ<sub>3</sub>:</b> Do the titles from Stack Overflow questions contain potential query keywords (i.e., technical terms) for code snippet search?

---

are annotated with *java* tag. In addition, we apply several other constraints—(1) each of the questions should have at least three answers (i.e., average answer count) with one answer being accepted as the *solution*, in order to ensure that the questions are answered substantially and successfully [148], and (2) the accepted answers should contain code like elements such as code snippets or code tokens so that API information can be extracted from them. We identify the code elements with the help of `<code>` tags in the HTML source of the answers (details in Section 7.2.2), and use *Jsoup*<sup>1</sup>, a popular Java library, for HTML parsing and content extraction.

We repeat the above steps, and construct another dataset by collecting 440K Q & A threads from one of our recent works [194]. This dataset is a superset of the above collection, and it contains more recent threads from Stack Overflow. We call it the *extended dataset* in the remaining sections of the exploratory study.

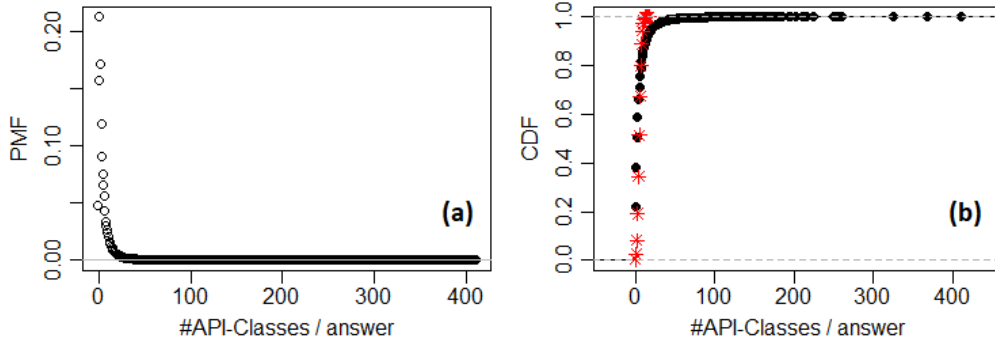
We collect a total of 3,809 Java API classes for our study from 19 packages of standard Java edition 7. While 2,912 classes are taken from 11 core Java packages<sup>2</sup>, the remaining classes have come from 8 non-core Java packages. The goal is to find out if these classes are referred to in Stack Overflow posts, and if yes, to what extent they are referred to. We first use Java Reflections [21], a runtime meta data analysis library, to collect the API classes from JDK 7, and then apply regular expressions on their fully qualified names for extracting the class name tokens. Table 7.1 shows class statistics of the 19 API packages selected for our investigation.

We also collect a set of 18,662 real life search queries from the Google search history of the first author over the last eight years, which are analysed to answer the third research question. Although the queries come from a single user, they contain a large vocabulary of 9,029 distinct natural language search keywords, and the vocabulary is built over a long period of time. Thus, a study using these queries can produce significant intuitions and help answer the third research question.

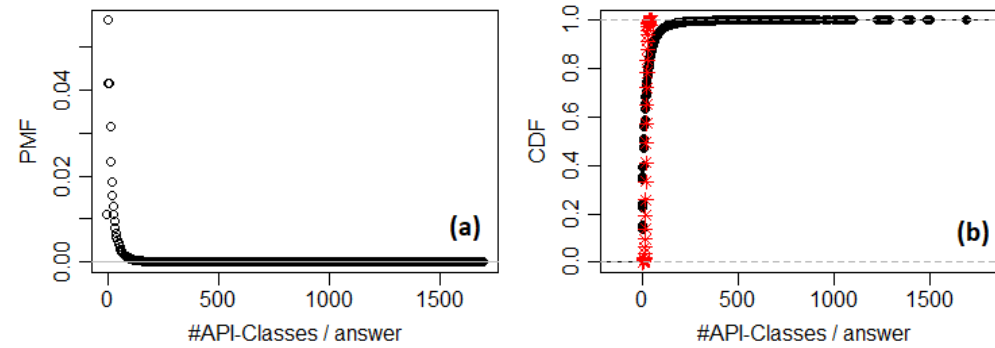
---

<sup>1</sup><https://jsoup.org/>

<sup>2</sup><https://goo.gl/A6gEqA>



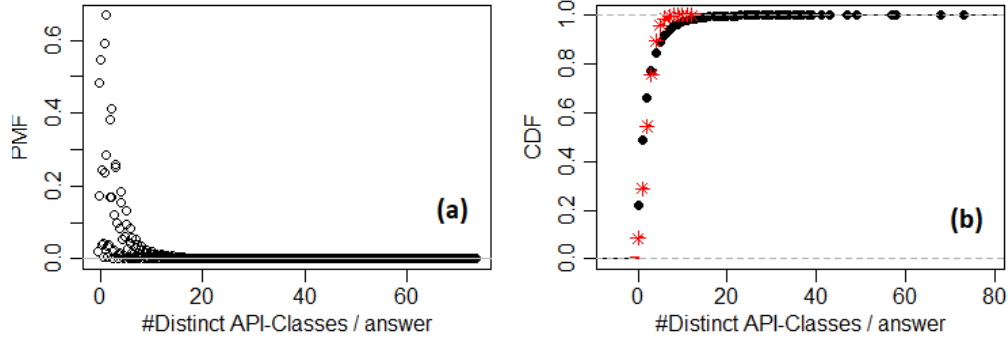
**Figure 7.2:** Frequency distribution for core API classes – (a) API frequency PMF, (b) API frequency CDF



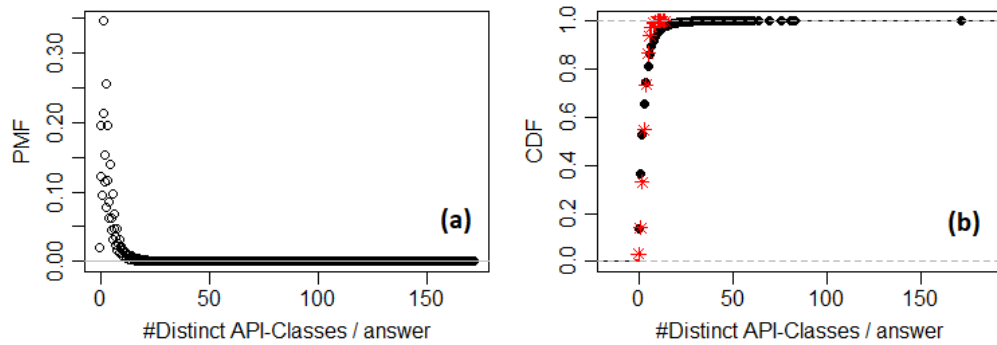
**Figure 7.3:** Frequency distribution for core and non-core API classes over the extended dataset – (a) API frequency PMF, (b) API frequency CDF

## 7.2.2 API Class Name Extraction

Several existing studies [42, 73, 211] extract code elements such as API packages, classes and methods from unstructured natural language texts (e.g., forum posts, mailing lists) using information retrieval (e.g., TF-IDF) and island parsing techniques. In the case of island parsing, they apply a set of regular expressions describing Java language specifications [89], and isolate the land (i.e., code elements) from water (i.e., free-form texts). We borrow their parsing technique [211], and apply it to the extraction of API elements from Stack Overflow posts. Since we are interested in the API classes only, we adopt a selective approach for identifying them in the post contents. We first isolate the code like sections from HTML source of each of the answers from Stack Overflow using `<code>` tags. Then we split the sections based on white spaces and punctuation marks, and collect the tokens having the camel-case notation of Java class (e.g., `HashSet`). According to the existing studies [73, 211], such parsing of code elements sometimes introduces false positives. Thus, we restrict our exploratory analysis to a closed set of 3,809 API classes from 19 Java packages (details in Table 7.1) to avoid false positives (e.g., camel-case tokens but not valid API classes).



**Figure 7.4:** Frequency distribution of unique API classes from core packages – (a) Distinct API frequency PMF, (b) Distinct API frequency CDF



**Figure 7.5:** Frequency distribution of unique API classes from core and non-core packages – (a) Distinct API frequency PMF, (b) Distinct API frequency CDF

### 7.2.3 Answering RQ<sub>1</sub>: Use of APIs in the accepted answers of Stack Overflow

Since our API suggestion technique exploits keyword-API associations from Stack Overflow, we investigate whether the accepted answers actually use certain API classes of interest in the first place. According to our investigation, out of 172,043 accepted answers, 136,796 (79.51%) answers refer to one or more Java API class-like tokens. About 61.02% of the answers actually use API classes from 11 core Java packages whereas 9.94% of them use the classes from 8 non-core packages as a part of their solution. We analyse the HTML contents from Stack Overflow answers with tool supports and then detect the *occurrences* of 3,809 standard API classes (Table 7.1) in each of the accepted answers using a closed-world assumption [211]. We then examine the statistical properties or distribution of such *API occurrence frequencies* (i.e., total appearances, unique appearances) and attempt to answer our first research question.

Fig. 7.2 shows (a) probability mass function (PMF) and (b) cumulative density function (CDF) for the total occurrences of API classes per SO answer where the API classes belong to the core Java API packages. Both density curves suggest that the frequency observations derive from a heavy-tailed distribution, and majority of the densities accumulate over a short frequency range. That is, most of the time only a limited number of API classes co-occur in each answer from Stack Overflow. The empirical CDF curve also closely matches with the theoretical CDF [5] (i.e., red dots in Fig. 7.2-(b)) of a Poisson distribution. Thus, we believe

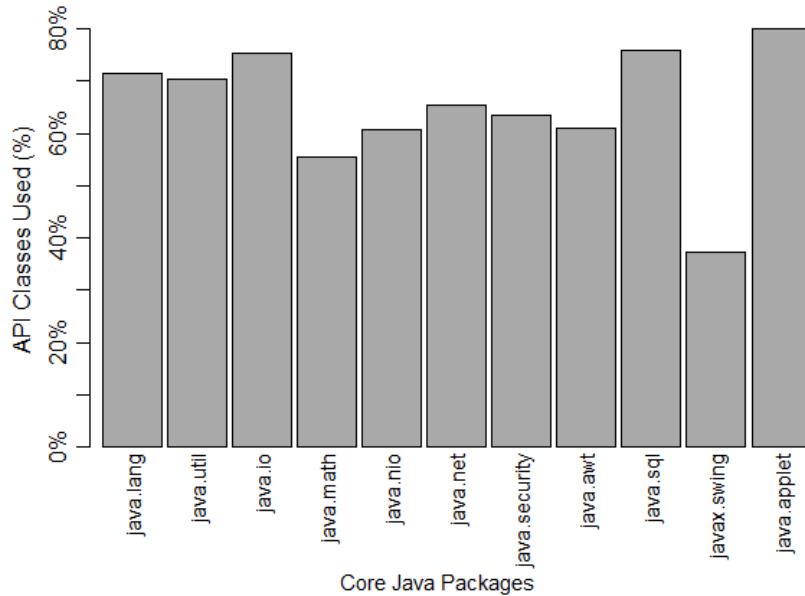
that the observations are probably taken from a Poisson distribution. We get a 95% confidence interval over [5.27, 5.37] for mean frequency,  $\lambda = 5.32$ , which suggests that the API classes from the core packages are referred to at least *five* times on average in each of the answers from Stack Overflow. We also get 10<sup>th</sup> quantile at frequency=2 and 97.5<sup>th</sup> quantile at frequency=10 which suggest that only 10% of the frequencies are below 3 and only 2.5% of the frequencies are above 10. When our investigation is repeated for non-core classes, we get a mean frequency,  $\lambda = 0.36$ , with 95% confidence interval over [0.35, 0.37]. When 11 core and 8 non-core packages are combined and employed against the *extended dataset*, we get a 95% confidence interval over [23.62, 23.87] for the mean frequency,  $\lambda=23.75$  with a similar distribution (i.e., Fig. 7.3). Fig. 7.4 shows density curves of the core API class occurrences per answer where only unique API classes are considered. These observations are also drawn from a heavy-tailed distribution. We get a 95% confidence interval over [2.35, 2.38] for the mean frequency,  $\lambda = 2.37$ , which suggests that at least two distinct classes are used on average in each answer. 30<sup>th</sup> quantile at frequency = 1 and 80<sup>th</sup> quantile at frequency = 4 suggest that 30% of the Stack Overflow answers refer to at least one API class whereas 20% of the answers refer to at least four distinct API classes from the core Java packages under our study. In the case of non-core classes, we get 90<sup>th</sup> quantile at frequency = 1, which suggests that their frequencies are negligible. When the same investigation is repeated with 19 (11 core + 8 non-core) packages against the *extended dataset*, we get a 95% confidence interval over [3.44, 3.46] for  $\lambda=3.45$  with a similar heavy tailed distribution (i.e., Fig. 7.5).

**Summary of RQ<sub>1</sub>:** At least **two** different API classes from the core Java packages are referred to in each of the **61%** accepted answers that are collected from Stack Overflow. These classes are mentioned at least **five** times on average in each answer. API classes from non-core packages are discussed in  $\approx$ **10%** of the answers. Furthermore, our observations derived from 172K answers are *similar* to that derived from an extended dataset of 440K answers from Stack Overflow.

#### 7.2.4 Answering RQ<sub>2</sub>: Coverage of API classes in the accepted answers from Stack Overflow Q & A site

Since our technique exploits inherent mapping between API classes in Stack Overflow answers and keywords from corresponding questions for API suggestion, we need to investigate if such answers actually use a significant portion of the API classes from the standard packages as a part of the solution. We thus identify the occurrences of the API classes from core and non-core packages (Table 7.1) in Stack Overflow answers, and determine the API coverage for these packages.

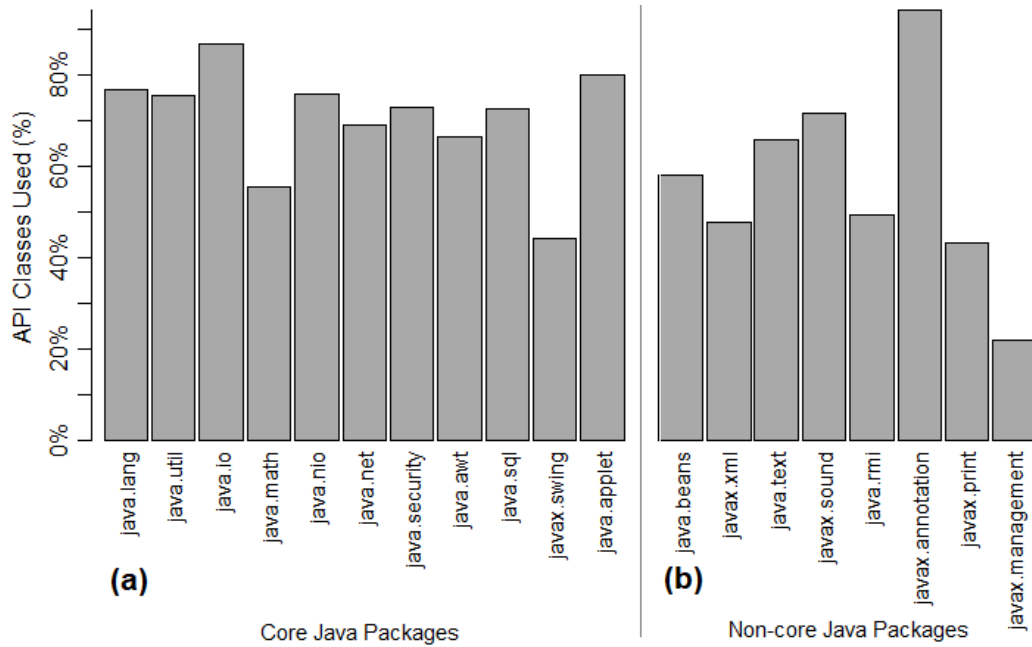
Fig. 7.6 shows the fraction of the API classes that are used in Stack Overflow answers for each of the 11 core packages under study. We note that at least 60% of the classes are used in Stack Overflow for nine out of 11 packages. The remaining two packages—`java.math` and `javax.swing` have 55.56% and 37.41% class coverage respectively. Among these nine packages, three large packages—`java.lang`, `java.util` and `java.io` have a class coverage over 70%. Thus, on average, 65% of the classes are mentioned at least once in Stack



**Figure 7.6:** Coverage of API classes from core packages by Stack Overflow answers

Overflow. In Fig. 7.7, when our investigations are repeated using 19 (11 core + 8 non-core) packages and an *extended dataset*, we get a 95% confidence interval over [56.11, 73.01] for mean coverage,  $\mu=64.56\%$  with a normal distribution. We note that at least 40% of the classes from seven non-core packages are used in Stack Overflow. The remaining package, `javax.management`, has a class coverage of  $\approx 20\%$ . Fig. 7.8 shows the fraction of Stack Overflow answers (under study) that use API classes from each of the core 11 packages. We see that classes from `java.lang` package are used in over 50% of the answers, which can be explained since the package contains a number of frequently used and basic classes such as `String`, `Integer`, `Method`, `Exception` and so on. Two packages— `java.util` and `java.awt` that focus on utility functions (e.g., unzip, pattern matching) and user interface controls (e.g., radio button, check box) respectively have a post coverage over 20%. We also note that classes from `java.io` and `javax.swing` packages are used in over 10% of the Stack Overflow answers, whereas the same statistic for the remaining six packages is less than 10%. When our investigations are repeated using 19 (11 core + 8 non-core) packages with the extended dataset, most of the above findings on core packages are reproduced, as shown in Fig. 7.9-(a). However, as in Fig. 7.9-(b)), we see that API classes from all eight non-core packages except `javax.xml` are used in less than 5% of the Stack Overflow answers under study. Thus, although a significant amount (e.g., 40%) of the classes from non-core packages are mentioned in Stack Overflow at least for once (i.e., Fig. 7.7-(b)), as a whole, they are less frequently discussed compared to the core classes. Such finding can also be explained by the highly specific functionalities (e.g., RMI, print) of the classes from non-core packages under study.

**Summary of RQ<sub>2</sub>:** On average, **65%** of the API classes from each of the 19 (core + non-core) Java packages are used in Stack Overflow accepted answers. Each of these packages is referred to (using their classes) by



**Figure 7.7:** Coverage of API classes from (a) core and (b) non-core packages by Stack Overflow answers (extended dataset)

**Table 7.3:** Keywords Intended for Code Search

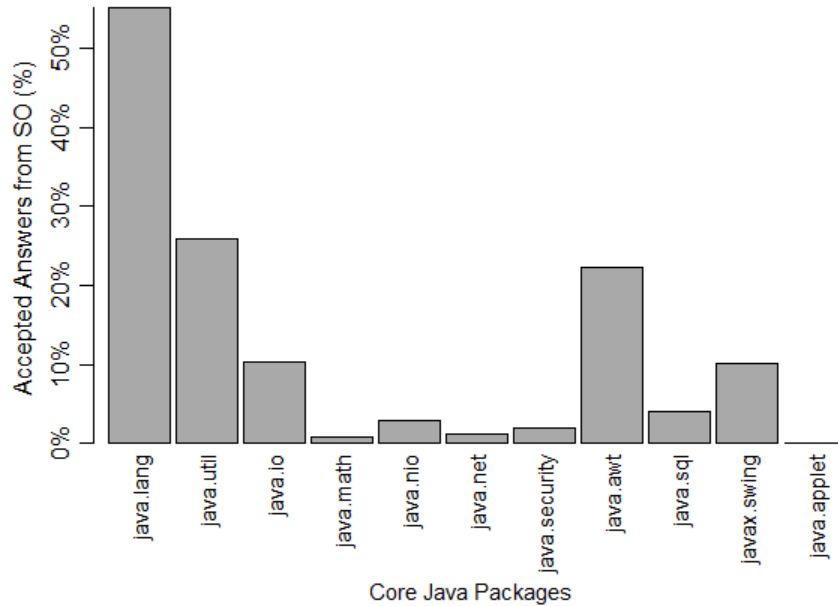
java	code	example
sql	server	file
string	mvc	web
add	type	lucene
android	table	programmatically

at least **10%** of the answers under our study. Such findings clearly suggest a significant presence of standard API classes in Stack Overflow posts, and thus, signal their high potential.

### 7.2.5 Answering RQ<sub>3</sub>: Presence of code search keywords in the title of questions from Stack Overflow

Our technique relies on the mapping between natural language terms from Stack Overflow questions and API classes from corresponding accepted answers for augmenting a code search query with relevant API classes. Thus, an investigation is warranted on whether keywords used for code search are present in the SO question texts or not. We are particularly interested in the title of a Stack Overflow question since it summarizes the technical requirement precisely using a few important words, and also resembles a search query. We analyse the titles of 172,043 Stack Overflow questions and 18,662 real life queries used for Google search (Section 7.2.1). Since we are interested in code related queries, we only select such queries that were intended for





**Figure 7.8:** Use of core API packages in the Stack Overflow answers

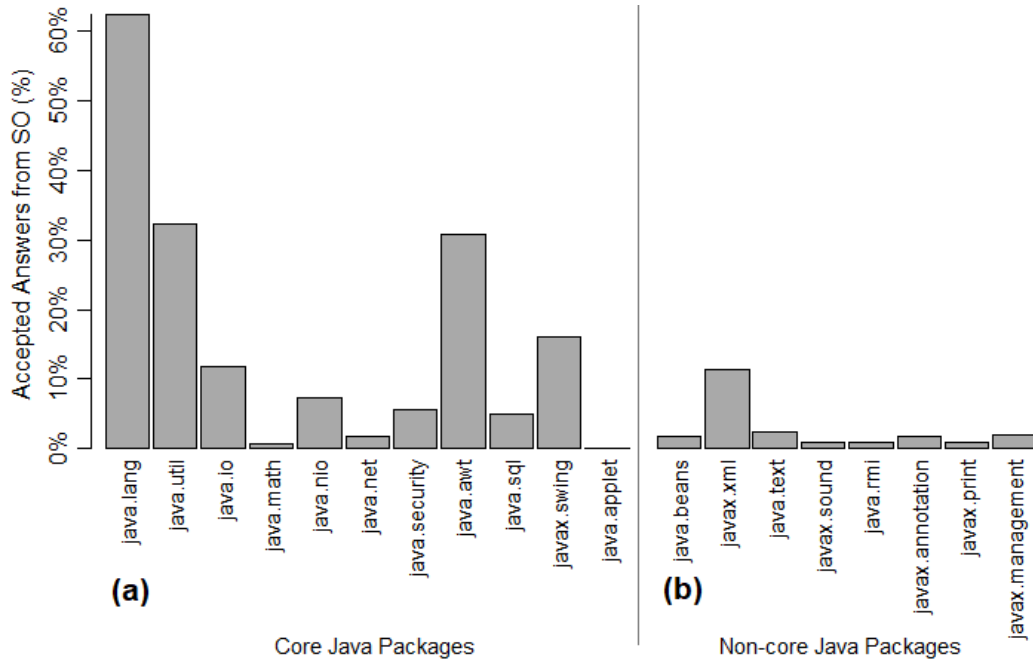
**Table 7.4:** Code Search Keywords Found in Tutorial Sites

Website	#Pages	#Terms	Source	Matched
Javatpoint	1,291	784	Title	20.54%
		10,099	Title+Body	60.12%
Tutorialspoint	2,219	1,292	Title	20.14%
		14,930	Title+Body	63.62%
<b>Stack Overflow</b>	172,043	20,391	Title	<b>69.22%</b>

**Matched**=Overlap between extracted terms and code search keywords

code search. Rahman et al. [205] recently used popular tags from Stack Overflow questions to separate code related queries from non-code queries that were submitted to a general-purpose search engine, Google. We use a subset of their selected tags (shown in Table 7.3) for identifying the code related queries. We discover 3,073 such queries from our query collection (Section 7.2.1) where the queries contain a total of 2,001 unique search keywords.

According to our analysis, 172,043 question titles contain 20,391 unique terms after performing natural language preprocessing (i.e., stop word removal, splitting and stemming). These terms match 69.22% of the keywords collected from our code search queries. Fig. 7.10 shows the fraction of the search keywords that match with the terms from Stack Overflow questions for the past eight years starting from 2008. On average, 62.69% of the code search keywords from each year match with Stack Overflow vocabulary derived from its question titles.



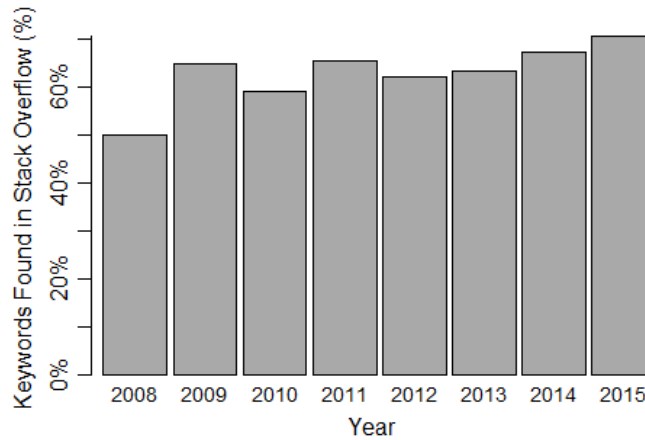
**Figure 7.9:** Use of (a) core and (b) non-core API packages in the Stack Overflow answers (extended dataset)

Fig. 7.11 shows (a) probability mass function, and (b) cumulative density function of keyword frequency in the question titles. We see that the density curve shows the central tendency like a normal curve (i.e., bell shaped curve), and the empirical CDF closely matches with the theoretical CDF (i.e., red curve) of a normal distribution with mean,  $\mu = 3.22$  and standard deviation,  $\sigma = 1.60$ . We also draw 172,043 random samples from a normal distribution with equal mean and standard deviation, and compare with the keyword frequencies. Our Kolmogorov-Smirnov test reported a  $p$ -value of  $2.2e-16 < 0.05$  which suggests that both sample sets belong to the same distribution. Thus, we believe that the keyword frequency observations come from a normal distribution. We get a mean frequency,  $\mu = 3.22$  with 95% confidence interval over  $[3.21, 3.23]$ , which suggests that each of the question titles from Stack Overflow contains at least three code search keywords on average. Furthermore, a recent query classification model that leverages Stack Overflow tags for separating code queries from non-code queries achieves a promising accuracy of 87% precision and 86% recall [205]. Such findings further suggest the potential of Stack Overflow vocabulary for improving the code search.

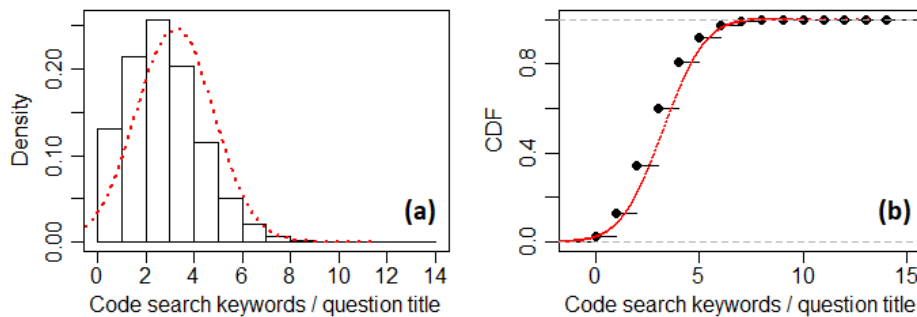
We also collect all the Q & A threads from two other popular tutorial sites—*Javatpoint*<sup>3</sup> and *Tutorialspoint*<sup>4</sup>, construct two *baseline vocabularies* from them, and then contrast with the vocabulary of Stack Overflow. Table 7.4 shows the statistics on downloaded pages and unique terms extracted from them. For example, Tutorialspoint has a total of 2,219 web pages, and they form a vocabulary of 14,930 unique terms when both title and body of the pages are considered. It encompasses various programming domains in-

<sup>3</sup><https://www.javatpoint.com>

<sup>4</sup><https://www.tutorialspoint.com>



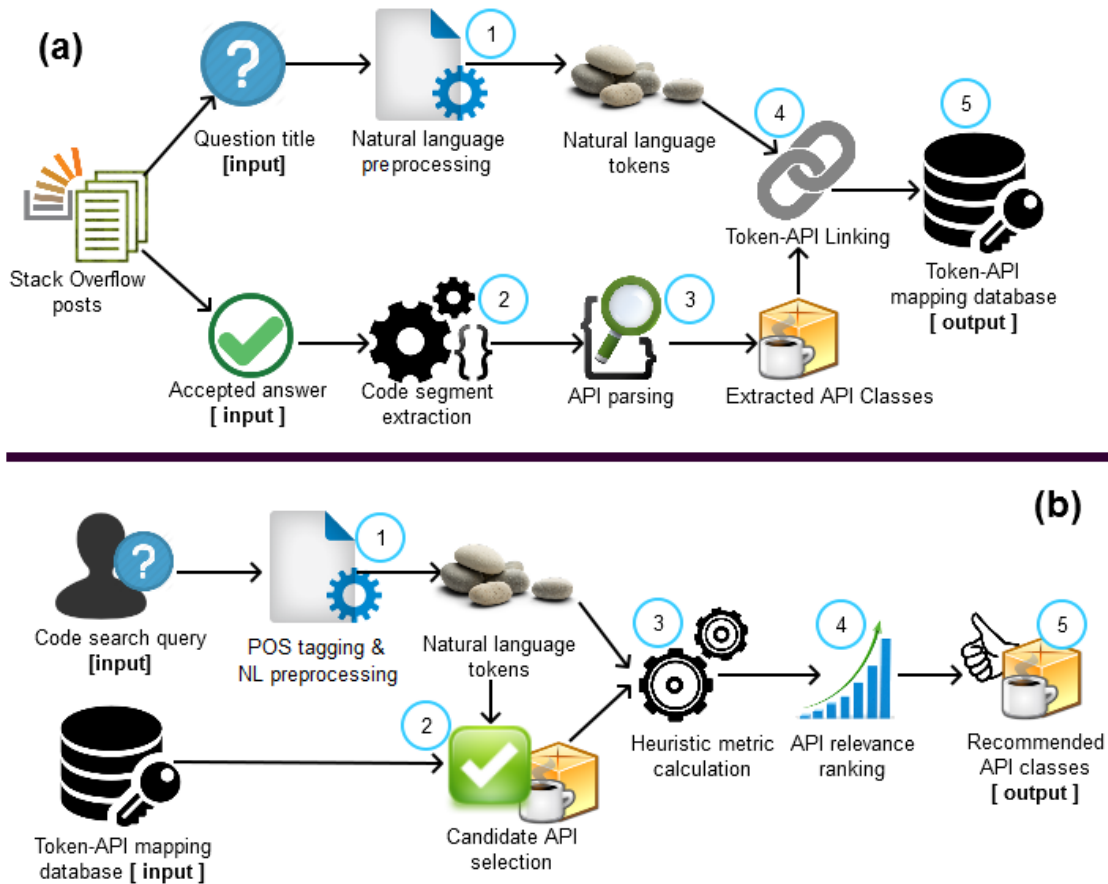
**Figure 7.10:** Coverage of keywords from the collected queries in Stack Overflow questions



**Figure 7.11:** Collected search query keywords in Stack Overflow– (a) Keyword frequency PMF (b) Keyword frequency CDF

cluding Java, C/C++, and C#. On the contrary, when titles from only Java related questions of Stack Overflow are considered, they form a vocabulary of 20K. We also note that terms from Tutorialspoint page titles match only  $\approx 20\%$  of the code search keywords. On the contrary, such matching ratio is 69% for Stack Overflow which is 237% higher. Surprisingly, when analysed from a granular perspective, Stack Overflow might not be better than these two sites. For example, titles from Javatpoint and Tutorialspoint provide 15.91% and 9.08% of search keywords as opposed to  $< 1.00\%$  by Stack Overflow when 1000 random pages are analysed. However, Stack Overflow offers (1) a nice combination of query terms (in the questions) and API classes (in the code snippets), and (2) a much larger collection of Q & A threads compared to Javatpoint and Tutorialspoint across various domains. Thus, it has a higher potential for assisting the developers in traditional code search.

**Summary of RQ<sub>3</sub>:** Each question title from Stack Overflow contains **three** potential keywords for code search on average. Term extracted from these titles match **69%** of the code search keywords produced in real life over the last **eight** years. Furthermore, vocabulary developed from Stack Overflow posts is *much larger* than that of any other available tutorial sites on the web.



**Figure 7.12:** Schematic diagram of the proposed query reformulation technique –RACK–(a) Construction of token-API mapping database, (b) Translation of a code search query into relevant API classes

### 7.3 RACK: Automated Query Reformulation for Internet-scale Code Search using Crowdsourced Knowledge

According to the exploratory study (Section 7.2), at least two API classes are used in each of the accepted answers of Stack Overflow, and about 65% of the API classes from the core packages are used in these answers. Besides, the titles from Stack Overflow questions are a major source of query keywords for code search. Such findings suggest that Stack Overflow might be a potential source not only for code search keywords but also for API classes relevant to them. Since we are interested in exploiting this keyword-API association from Stack Overflow questions and answers for API suggestion (i.e., for query reformulation), we need a technique that stores such associations, mines them automatically, and then recommends the most relevant APIs. Thus, our proposed technique has two major steps – (a) Construction of token-API mapping database, and (b) Recommendation of relevant API classes for a code search query which is written in natural language (a.k.a., NL query). Fig. 7.12 shows the schematic diagram of our proposed technique–RACK– for automated query reformulation with relevant API classes for Internet-scale code search.

### 7.3.1 Construction of NL Token-API Mapping Database

Since our technique relies on keyword-API associations from Stack Overflow, we need to extract and store such associations for quick access. In Stack Overflow, each question describes a technical requirement such as “*how to send an email in Java?*” The corresponding answer offers a solution containing code example(s) that refer(s) to one or more API classes (e.g., `MimeMessage`, `Transport`). We capture both the requirement and API classes carefully, and exploit their semantic association for the development of token-API mapping database. Since the title summarizes a question using a few but important words, we only use the titles from the questions. Acceptance of an answer by the person who posted the question indicates that the answer actually meets the requirement in the question. Thus, we consider only the accepted answers from the answer collection for our analysis. The construction of the mapping database has several steps as follows:

**Token Extraction from Titles:** We collect title(s) from each of the questions, and apply standard natural language pre-processing steps such as stop word removal, splitting and stemming on them (Step 1, Fig. 7.12-(a)). Stop words are the frequently used words (e.g., *the*, *and*, *some*) that carry very little meaning for a sentence. We use a stop word list [25] hosted by Google for the stop word removal step. The splitting step splits each word containing any punctuation mark (e.g., `.,?;!;`), and transforms it into a list of words. Finally, the stemming step extracts the root of each of the words (e.g., “send” from “sending”) from the list, where Snowball stemmer [176, 246] is used. Thus, we extract a set of unique and stemmed words that collectively convey the meaning of the question title, and we consider them as the “tokens” from the title of a question from Stack Overflow. Finally, our database ended up with a total of 19,783 unique NL terms.

**API Class Extraction:** We collect the accepted answer for each of our selected questions, and parse their HTML source using Jsoup parser [14] for code segments (Step 2, 3, Fig. 7.12-(a)). We extract all `<code>` and `<pre>` tags from the source content as they generally contain code segments [198]. It should be noted that code segments may sometimes be demarcated by other tags or no tag at all. However, identification of such code segments is challenging and often prone to false-positives. Thus, we restrict our analysis to contents inside `<code>` tags and `<pre>` for code segment collection from Stack Overflow. We split each of the segments based on punctuation marks and white spaces, and discard the programming keywords. Existing studies [42, 211] apply island parsing for API method or class extraction where they use a set of regular expressions. Similarly, we use a regular expression for Java class [89], and extract the API class tokens having a camel case notation. Thus, we collect a set of unique API classes from each of the accepted answers. The API classes (e.g., `String`, `Integer`, `Double`) from `java.lang` package are mostly generic and frequently used in the code, which is also supported by our RQ<sub>2</sub>. Hence, we also avoid all the API classes from this package during our API extraction from Stack Overflow answers.

**Token-API Linking:** Natural language tokens from a question title hint about the technical requirement described in the question, and API names from the accepted answer represent the relevant APIs that can meet such requirement. Thus, the programming Q & A site—Stack Overflow— inherently provides an important semantic association between a list of tokens and a list of APIs. For instance, our technique generates a list

of natural language tokens—*{generat, md5, hash}*— and an API token—`MessageDigest`— from the showcase example on MD5 hash (Fig. 7.1). We capture such associations from 126,567 Stack Overflow question and accepted answer pairs, and store them in a relational database (Step 4, 5, Fig. 7.12-(a)) for relevant API recommendation for any code search query.

### 7.3.2 API Relevance Ranking & Reformulation of the NL-Query

In the token-API mapping database, each NL token (or term) associates with different APIs, and each API class associates with a number of NL tokens. Thus, we need a technique that carefully analyses such associations, identifies the candidate APIs, and then recommends the most relevant ones from them for a given query. It should be noted that we do not apply the traditional association rule mining [264]. Our investigations using the constructed database (Section 7.3.1) report that frequencies of co-occurrence between NL terms and API classes in Stack Overflow posts are not sufficient enough to form association rules for all queries. The API class ranking and recommendation targeting our query reformulation for code search involve several steps as follows:

#### Identification of Keyword Context

In natural language processing, the context of a word refers to the list of other words that co-occur with that word in the same phrase, same sentence or even the same paragraph [100]. Co-occurring words complement the semantics of one another [153]. Yuan et al. [272] analyse programming posts and tags from Stack Overflow Q & A site, and use word context for determining semantic similarity between any two software-specific words. In this research, we identify the words that co-occur with each query keyword in the thousands of question titles from Stack Overflow. For each keyword, we refer to these co-occurring words as its *context*. We then opportunistically use these contextual words for estimating semantic relevance between any two keywords.

#### Candidate API Selection

In order to collect candidate APIs for a NL query, we employ three different heuristics. These heuristics consider not only the association between query keywords and APIs but also the coherence among the APIs themselves. Thus, the key idea is to identify such programming APIs as candidates that are not only likely for the query keywords but also functionally consistent to one another.

**Keyword-API Co-occurrence (KAC):** Stack Overflow discusses thousands of programming problems, and these discussions contain both natural language texts (i.e., keywords) and reference to a number of APIs. According to our observation, several keywords might co-occur with a particular API and a particular keyword might co-occur with several APIs across different programming solutions. This co-occurrence generally takes place either by chance or due to semantic relevance. Thus, if carefully analysed, such co-occurrences could be a potential source for semantic association between keywords and APIs. We capture these co-occurrences (i.e., associations) between keywords from question titles and APIs from accepted answers, discard the random

associations using a heuristic threshold ( $\delta$ ), and then collect the top API classes ( $L_{KAC}[K_i]$ ) for each keyword ( $K_i$ ) that co-occurred most frequently with the keyword at Stack Overflow.

$$L_{KAC}[K_i] = \{A_j \mid A_j \in A \wedge \text{rank}_{freq}(K_i \rightarrow A_j) \leq \delta\} \quad (7.1)$$

Here,  $K_i \rightarrow A_j$  denotes the association between a keyword  $K_i$  and an API class  $A_j$ ,  $\text{rank}_{freq}$  returns rank of the association from the ranked list based on association frequency, and  $\delta$  is a heuristic rank threshold. In our research, we consider top ten (i.e.,  $\delta = 10$ ) APIs as candidates for each keyword, which is carefully chosen based on iterative experiments on our dataset (see RQ<sub>7</sub> for details).

**Keyword Pair-API Co-occurrence (KPAC):** While frequent co-occurrences of APIs with a query keyword are a good indication of their relevance to the query, they might also fall short due to the fact that the query might contain more than one keyword. That is, API classes relevant to (i.e., frequently co-occurred with) one keyword might not be relevant to other keywords from the query. Thus, API classes that are simultaneously relevant to multiple keywords should be selected as candidates. We consider  ${}^nC_2$  keyword pairs from  $n$  keywords of a query using combination theory, and identify such APIs that frequently co-occur with both keywords from each pair in the same context (e.g., same Q & A thread). Suppose,  $K_i$  and  $K_j$  are two keywords, and they form one of the  ${}^nC_2$  keyword pairs from the query. Now, the candidate API classes ( $L_{KPAC}[K_i, K_j]$ ) are relevant if they occur in an accepted answer of Stack Overflow whereas both keywords appear in the corresponding question title. We select such relevant candidates as follows:

$$L_{KPAC}[K_i, K_j] = \{A_m \mid A_m \in A \wedge \text{rank}_{freq}((K_i, K_j) \rightarrow A_m) \leq \delta\} \quad (7.2)$$

Here  $(K_i, K_j) \rightarrow A_m$  denotes the association between keyword pair  $(K_i, K_j)$  from a question title and API class  $A_m$  from the corresponding accepted answer of Stack Overflow. We capture top ten (i.e.,  $\delta = 10$ ) such co-occurrences for KPAC heuristic, and the detailed justification for this choice can be found in RQ<sub>7</sub>. We determine the association based on their co-occurrences in the same set of documents. In this case, each question-answer thread from Stack Overflow is considered as a document. While co-occurrences of keyword triples with APIs could also be considered for API candidacy, existing IR-based studies report that phrases of two words are more effective as a semantic unit (e.g., “*chat room*”) rather than the triples (e.g., “*find chat room*”) [153, 191].

**Keyword-Keyword Coherence (KKC):** The two heuristics above determine relevant API candidacy based on the co-occurrence between query keywords and APIs in the same document. That is, multiple keywords from the query are also warranted to co-occur in the same document. However, such co-occurrences might not always happen, and yet the keywords could be semantically related to one another (i.e., co-occurred in the query). More importantly, the candidate APIs should be relevant to multiple keywords that do not co-occur. Yuan et al. [272] determine semantic similarity between any two software specific words by using their contexts from Stack Overflow questions and answers. We adapt their technique for identifying coherent keyword pairs which might not co-occur. The goal is to collect candidate APIs relevant to these pairs based on

their coherence. We (1) develop a context ( $C_i$ ) for each of the  $n$  query keywords by collecting its co-occurring words from thousands of question titles from Stack Overflow, (2) determine semantic similarity for each of the  $nC_2$  keyword pairs based on their context derived from Stack Overflow, and (3) use these measures to identify the coherent pairs and then to collect the functionally coherent APIs for them. At the end of this step, we have a set of candidate APIs for each of the coherent keyword pairs.

Suppose, two query keywords  $K_i$  and  $K_j$  have context word list  $C_i$  and  $C_j$  respectively. Now, the candidate APIs ( $L_{KKC}$ ) that are relevant to both query keywords and functionally consistent with one another can be selected as follows:

$$L_{KKC}[K_i, K_j] = \{L[K_i] \cap L[K_j] \mid \cos(C_i, C_j) > \gamma\} \quad (7.3)$$

Here,  $\cos(C_i, C_j)$  denotes the *cosine similarity* [198] between two context lists—  $C_i$  and  $C_j$ , and  $\gamma$  is the similarity threshold. We consider  $\gamma = 0$  in this work based on iterative experiments on our dataset (see RQ<sub>7</sub> for the detailed justification).  $L[K_i]$  and  $L[K_j]$  are top frequent APIs for the two keywords—  $K_i$  and  $K_j$  where  $K_i$  and  $K_j$  might not co-occur in the same question title. Thus,  $L_{KKC}[K_i, K_j]$  contains such APIs that are relevant to both keywords (i.e., co-occurred with them in Stack Overflow answers) and functionally consistent with one another. Since the candidate APIs co-occur with the keywords from each coherent pair (i.e., semantically similar,  $\gamma > 0$ ) in different contexts, they are also likely to be *coherent* for the programming task at hand. Such coherence often could be explained in terms of the dependencies among the API classes.

### API Relevance Ranking Algorithm

Fig. 7.12-(b) shows the schematic diagram, and Algorithm 8 shows the pseudo code of our API relevance ranking algorithm—RACK. Once a search query is submitted, we (1) perform Part-of-Speech (POS) tagging on the query for extracting the meaningful words such as nouns and verbs [59, 259], and (2) apply standard natural language preprocessing (i.e., stop word removal, splitting, and stemming) on them to extract the stemmed words (Lines 3–4, Algorithm 8). For example, the query—“*html parser in Java*” turns into three keywords—‘*html*’, ‘*parser*’ and ‘*java*’ at the end of the above step. We then apply our three heuristics— $KAC$ ,  $KPAC$  and  $KKC$ —on the stemmed keywords, and collect candidate APIs from the token-API linking database (Step 2, Fig. 7.12-(b), Lines 5–8, Algorithm 8). The candidate APIs are selected based on not only their co-occurrence with the query keywords but also the coherence (i.e., functional consistency) among themselves. We then use the following metrics (i.e., derived from the above heuristics) to estimate the relevance of the candidate API classes for the query.

**API Co-occurrence Likelihood** estimates the probability of co-occurrence of a candidate API ( $A_j$ ) with one ( $K_i$ ) or more ( $K_i, K_j$ ) keywords from the search query. It considers the rank of the API in the ranked list based on keyword-API co-occurrence frequency (i.e.,  $KAC$  and  $KPAC$ ) and the size of the list, and then provides a normalized score (on the scale from 0 to 1) as follows:



---

**Algorithm 8** API Relevance Ranking using the Proposed Heuristics

---

```
1: procedure RACK( $Q$ ) ▷  $Q$ : natural language query for code search
2:    $R \leftarrow \{\}$  ▷ list of API classes relevant to  $Q$ 
3:   ▷ collecting keywords using POS tagging and NL preprocessing
4:    $K \leftarrow \text{preprocess}(\text{collectNounVerbs}(Q))$ 
5:   ▷ collecting candidate API classes
6:    $L_{KAC} \leftarrow \text{getKACList}(K)$ 
7:    $L_{KPAC} \leftarrow \text{getKPACList}(K)$ 
8:    $L_{KKC} \leftarrow \text{getKKCList}(K)$ 
9:   ▷ estimating relevance of the candidate APIs
10:  for Keyword  $K_i \in K$  do
11:    for APIClass  $A_j \in L_{KAC}[K_i]$  do
12:      ▷ relevance of an API with single keyword
13:       $S_{KAC} \leftarrow \text{getKACScore}(A_j, L_{KAC}[K_i])$ 
14:       $R_{KAC}[A_j].score \leftarrow R_{KAC}[A_j].score + S_{KAC}$ 
15:    end for
16:  end for
17:  for Keyword  $K_i, K_j \in K$  do
18:    ▷ relevance of an API with multiple keywords
19:    for APIClass  $A_j \in L_{KPAC}[K_i, K_j]$  do
20:       $S_{KPAC} \leftarrow \text{getKPACScore}(A_j, L_{KPAC}[K_i, K_j])$ 
21:       $R_{KPAC}[A_j].score \leftarrow R_{KPAC}[A_j].score + S_{KPAC}$ 
22:    end for
23:    ▷ coherence of an API with other candidate APIs
24:     $C_i \leftarrow \text{getContextList}(K_i)$ 
25:     $C_j \leftarrow \text{getContextList}(K_j)$ 
26:     $S_{KKC} \leftarrow \text{getKKCScore}(C_i, C_j)$ 
27:    for APIClass  $A_j \in L_{KKC}[K_i, K_j]$  do
28:       $R_{KKC}[A_j].score \leftarrow R_{KKC}[A_j].score + S_{KKC}$ 
29:    end for
30:  end for
31:  ▷ ranking of the API classes using their normalized scores and relative weights
32:  for APIClass  $A_j \in \{R_{KAC}, R_{KPAC}, R_{KKC}\}$  do
33:     $R[A_j] \leftarrow \max(\alpha \times R_{KAC}[A_j], \beta \times R_{KPAC}[A_j], (1 - \alpha - \beta) \times R_{KKC}[A_j])$ 
34:  end for
35:  return  $\text{sortByScore}(R)$ 
36: end procedure
```

---

$$S_{KAC}(A_j, K_i) = 1 - \frac{\text{rank}(A_j, \text{sortByFreq}(L[K_i]))}{|L_{KAC}[K_i]|} \quad (7.4)$$

$$S_{KPAC}(A_j, K_i, K_j) = 1 - \frac{\text{rank}(A_j, \text{sortByFreq}(L_{KPAC}[K_i, K_j]))}{|L_{KPAC}[K_i, K_j]|} \quad (7.5)$$

Here,  $S_{KAC}$  and  $S_{KPAC}$  denote the API co-occurrence likelihood estimates, and they range from 0 (i.e., not likely at all for the keywords) to 1 (i.e., very much likely for the keywords). The more likely an API is for the keywords, the more relevant it is for the query. This approach might also encourage the common API classes (e.g., `List`, `String`) that are often used with most programming tasks. Such APIs might not be helpful for relevant code snippet search. We thus apply appropriate filters and thresholds to avoid such noisy items.

**API Coherence** estimates the coherence of an API ( $A_j$ ) with other candidate APIs for a query. Since the query targets a particular programming task (e.g., “*parsing the HTML source*”), the suggested APIs should be logically consistent with one another. One way to heuristically determine such coherence is to exploit the semantic relevance among the corresponding keywords that co-occurred with that API ( $A_j$ ). The underlying idea is that if two keywords are semantically similar, their co-occurred API sets could also be logically consistent with each other. We thus determine semantic similarity between any two keywords ( $K_i, K_j$ ) from the query using their context lists ( $C_i, C_j$ ) [272], and then propagate that measure to each of their candidate API classes ( $A_j$ ) that co-occurred with both of the keywords (i.e.,  $KKC$ ) as follows:

$$S_{KKC}(A_j, K_i, K_j) = \text{cos}(C_i, C_j) \mid (K_i \rightarrow A_j) \wedge (K_j \rightarrow A_j) \quad (7.6)$$

Here,  $S_{KKC}$  denotes the API Coherence estimate, and it ranges from 0 (i.e., not relevant at all with multiple keywords) to 1 (i.e., highly relevant). It should be noted that each candidate,  $A_j$ , comes from  $L[K_i]$  or  $L[K_j]$ , i.e., the API is already relevant to each of  $K_i$  and  $K_j$  in their corresponding contexts.  $S_{KKC}$  investigates how similar those contexts are, and thus heuristically estimates the coherence between the APIs from these contexts.

We first estimate *API Co-occurrence Likelihood* of each of the candidate APIs that suggests the likeliness of the API for one or more keywords from the given query (Lines 9–22, Algorithm 8). Then we determine *API Coherence* for each candidate API that suggests coherence of the API with other candidate APIs for the query (Lines 23–30). Once all metrics of each candidate are calculated (Step 3, Fig. 7.12-(b)), only the maximum score is taken into consideration where appropriate weights— $\alpha$ ,  $\beta$  and  $(1 - \alpha - \beta)$ —are applied (Lines 31–34, Algorithm 8). These weights control how two of our above dimensions—*co-occurrence* and *coherence*—affect the final relevance ranking of the candidates. We consider a heuristic value of 0.325 for  $\alpha$  and a value of 0.575 for  $\beta$ , and the detailed weight selection method is discussed in Section 7.4.9. The candidates are then ranked based on their final scores, and Top-K API classes from the ranked list are returned as API recommendation (Line 35, Algorithm 8, Step 4, 5, Fig. 7.12-(b)). Such API classes are then used for NL-query reformulation.

**Table 7.5:** An Example of Query Reformulation using RACK

	html	$S_{KAC}$	parser	$S_{KAC}$	java	$S_{KAC}$
<b>KAC</b>	Document	1.00	Document	1.00	Object	1.00
	Jsoup	0.80	Element	0.80	ArrayList	0.80
	Element	0.60	File	0.60	File	0.60
	Elements	0.40	IOException	0.40	Class	0.40
	IOException	0.20	Node	0.20	IOException	0.20
	(html, parser)	$S_{KPAC}$	(html, java)	$S_{KPAC}$	(parser, java)	$S_{KPAC}$
<b>KPAC</b>	Document	1.00	Document	1.00	Document	1.00
	Jsoup	0.80	Jsoup	0.80	Element	0.80
	Element	0.60	Element	0.60	File	0.60
	Elements	0.40	IOException	0.40	DocumentBuilder	0.40
	Parser	0.20	Elements	0.20	DocumentBuilderFactory	0.20
	(html, parser)	$S_{KKC}$	(html, java)	$S_{KKC}$	(parser, java)	$S_{KKC}$
<b>KKC</b>	Document	0.42	IOException	0.28	File	0.20
	Element	0.42	File	0.28	IOException	0.20
	IOException	0.42				
	File	0.42				
	ArrayList	0.42				
	Initial Query		Reformulated Query	Suggested API		Score
<b>RACK</b>	$Q = \text{"HTML parser in Java"}$		$Q' = \{\text{Document, Element, File, IOException, Jsoup}\} + Q$	Document	0.79	
				Element	0.69	
				File	0.69	
				IOException	0.52	
				Jsoup	0.50	

**Working Example:** Table 7.5 shows a working example of how our proposed query reformulation technique –RACK– works. Here we reformulate our natural language query–“*HTML parser in Java*”–into relevant API classes. We first apply *KAC* heuristic, and collect the Top-5 (i.e.,  $\delta = 5$ ) candidate APIs for each of the three keywords–‘*html*’, ‘*parser*’ and ‘*java*’– based on co-occurrence frequencies of the candidates with the keywords. We also repeat the same step for each of the three (i.e.,  ${}^3C_2$ ) keyword pairs–(*html*, *parser*), (*html*, *java*) and (*parser*, *java*) by applying our *KPAC* heuristic. Then we estimate *co-occurrence likelihood* (with the keywords and keyword pairs) of each of the candidate APIs. For example, `Document` has a maximum likelihood of 1.00 among the candidates not only for the single search keyword but also for the keyword pairs. We then determine *coherence* of each candidate API (with other candidates) based on semantic relevance among the above three keyword pairs. For example, ‘*html*’ and ‘*parser*’ have a semantic relevance of 0.42 between them (on the scale from 0 to 1) based on their contexts from Stack Overflow questions and answers, and they have several common candidates such as `Document`, `Element` and `File`. Since the two keywords are semantically relevant, their relevance score (i.e., 0.42,  $S_{KKC}$ ) is propagated to their shared candidate APIs as a proxy to the coherence among the candidates. We then gather all scores for each candidate, choose the best score, and finally get a ranked list. From the recommended list, we see that `Document`, `Element` and `Jsoup` are highly relevant APIs from *Jsoup library* for the given NL-query. Our technique returns such a list of relevant API classes as the reformulation to an original NL query.

## 7.4 Experiment

One of the most effective ways to evaluate a technique that suggests relevant API classes or methods for a query is to check their conformance with the gold set APIs of the query. Since the suggested APIs could be used to reformulate the initial query (i.e., using natural language), the quality of the automatically reformulated query could be another performance indicator for the technique. We evaluate our technique using 175 code search queries, their goldset APIs and their relevant code segments (i.e., implementing the tasks in the query) collected from three programming tutorial sites. We determine the performance of our technique using six appropriate metrics from the literature. Then we compare with two variants of the state-of-the-art technique on API recommendation [243] and a popular code search engine–*Lucene* [98]–for validating our performance. We answer eight research questions with our experiments as shown in Table 7.6.

### 7.4.1 Experimental Dataset

**Data Collection:** We collect 175 code search queries for our experiment from three Java tutorial sites–KodeJava [15], JavaDB [13] and Java2s [11]. These sites discuss hundreds of programming tasks that involve the usage of different API classes from the standard Java API libraries. Each of these task descriptions generally has three parts–(1) a title (i.e., question) for the task, (2) one or more code snippets (i.e., answer), and (3) an associated prose explaining the code. The title summarizes a programming task (e.g., “*How do I*

**Table 7.6:** Research Questions Answered using our Experiment

---

**Research Questions on API Suggestion**

---

**RQ<sub>4</sub>:** How does the proposed technique –RACK– perform in recommending relevant APIs for a code search query?

---

**RQ<sub>5</sub>:** How effective are the proposed heuristics–*KAC*, *KPAC* and *KKC*–in capturing the relevant APIs for a query?

---

**RQ<sub>6</sub>:** Does an appropriate subset of the query keywords perform better than the whole query in retrieving the relevant APIs?

---

**RQ<sub>7</sub>:** How do the heuristic weights (i.e.,  $\alpha$ ,  $\beta$ ) and threshold settings (i.e.,  $\gamma$ ,  $\delta$ ) influence the performance of our technique?

---

**RQ<sub>8</sub>:** Can RACK outperform the state-of-the-art techniques in recommending relevant APIs for a given set of natural language queries?

---

---

**Research Questions on Query Reformulation**

---

**RQ<sub>9</sub>:** Can RACK significantly improve the natural language queries in terms of relevant code retrieval performance?

---

**RQ<sub>10</sub>:** Can RACK outperform the state-of-the-art technique in improving the natural language queries for code search?

---

**RQ<sub>11</sub>:** How does RACK perform compared to the popular web search engines (e.g., Google) and code search engines (e.g., GitHub code search)?

---

*decompress a GZip file in Java?*") using natural language texts. It generally uses a few pertinent keywords (e.g., "decompress", "GZip"), and also often resembles a query for code search (Section 7.2.5). We thus consider such titles from the tutorial sites as the code search queries, and use them for our experiment in this research.

**Gold Set Development:** The prose explaining the code often refers to one or more APIs (e.g., `GZipOutputStream`, `FileOutputStream`) from the code snippet(s) that are found to be essential for the task. In other words, such APIs can be considered as the most relevant ones (i.e., vital) for the target programming task. We collect such APIs from the prose against each of the task titles (i.e., code search queries) from our dataset, and develop a gold set—*API-goldset*—for the experiment. Since relevance of the APIs is determined based on working code examples and their associated prose from the publicly available and popular tutorial sites, the subjectivity associated with the relevance of the collected APIs is minimized [63]. We also collect the code segments verbatim that implement each of the selected tasks (i.e., our queries) from these tutorial sites, and develop another gold set—*Code-goldset*—for our experiments. Our goals are to (1) compare our queries containing the suggested API classes with the baseline queries containing only NL keywords and (2) compare our queries with the reformulated queries by the state-of-the-art techniques on API recommendation [168, 243, 274].

**Corpus Preparation:** We evaluate not only the API recommendation performance of RACK but also the retrieval performance of its reformulated queries. We collect relevant code snippets (i.e., ground truth) for each of our 175 search queries from the above tutorial sites, and develop a corpus. It should be noted that each query-code snippet pair comes from the same Q & A thread from the tutorial sites. However, this approach leaves us with a corpus of 175 documents which do not represent a real world corpus. We thus extend our code corpus by adding more code snippets from one of our earlier works [184], and this provided a corpus containing 4,170 (175+3,995) code snippets. It should be noted that the additional 3,995 code snippets were carefully collected from hundreds of open source projects hosted at GitHub (see [184] for details). This corpus is referred to as *4K-Corpus* throughout the later sections in the chapter.

We also develop two other corpora containing 256,754 (175+256,399) and 769,244 (175+769,069) documents respectively. They are referred to as *256K-Corpus* and *769K-Corpus* in the rest of the sections. These corpus documents are Java classes extracted from an internet-scale and well-established dataset—*IJaDataset* [118, 143, 236]. The dataset was constructed using 24,666 real world Java projects across various domains, and they were collected from SourceForge<sup>5</sup> and Google Code<sup>6</sup> repositories. We analyse 1,500,000 Java source files from the dataset, and discard the ones with a size greater than 3KB. 95% of our ground truth code segments have a size less than 3KB. The goal was to avoid the large and potentially noisy code snippets in the corpus. Given the large size (i.e., 769K documents) and cross-domain nature of the collected projects, our corpora are thus likely to represent a real world code search scenario.

---

<sup>5</sup><https://sourceforge.net/>

<sup>6</sup><https://code.google.com/>

We consider each of these code snippets from all three corpora as an individual document, apply standard natural language preprocessing (i.e., token splitting, stop word removal, programming keyword removal) to them, and then index the corpus documents using *Apache Lucene*<sup>7</sup>, a search engine widely used by the relevant literature [98, 120, 163]. The indexed corpus is then used to determine the retrieval performance of the initial and reformulated queries for code search.

**Replication:** All the experimental data, associated tools and implementations are hosted online [201] for replication or third party reuse.

## 7.4.2 Performance Metrics

We choose six performance metrics for the evaluation and validation of our technique that are widely adopted by relevant literature [63, 152, 243]. Two of them are related to recommendation systems whereas the other four metrics are widely popular in the information retrieval domain.

**Top-K Accuracy/Hit@K:** It refers to the percentage of the search queries for each of which at least one item (e.g., API class, code segment) is correctly returned within the Top-K results by a recommendation technique. It is also called Hit@K [239, 250]. Top-K Accuracy of a technique can be defined as follows:

$$Top\text{-}K\text{ Accuracy}(Q) = \frac{\sum_{q \in Q} isCorrect(q, K)}{|Q|} \%$$

Here,  $isCorrect(q, K)$  returns a value 1 if there exists at least one correct API class (i.e., from the API-goldset) or one correct code segment (i.e., implements the task in query) in the Top-K returned results, and returns 0 otherwise.  $Q$  denotes the set of all search queries used in the experiment. Although Top-K Accuracy and Hit@K are used interchangeably in the literature [239, 249], we use Hit@K to denote recommendation accuracy in the remaining sections for the sake of consistency.

**Mean Reciprocal Rank@K (MRR@K):** Reciprocal rank@K refers to the multiplicative inverse of the rank (i.e.,  $1/rank(q, K)$ ,  $q \in Q$ ) of the first relevant API class or code segment in the Top-K results returned by a technique [220, 276]. Mean Reciprocal Rank@K (MRR@K) averages such measures for all search queries ( $\forall q \in Q$ ) in the dataset. It can be defined as follows:

$$MRR@K(Q) = \frac{1}{|Q|} \sum_{q \in Q} \frac{1}{rank(q, K)}$$

Here,  $rank(q, K)$  returns the rank of the first correct API or the correct code segment from a ranked list of size K. If no correct API class or code segment is found within the Top-K positions, then  $rank(q, K)$  returns  $\infty$ . On the contrary, it returns 1 for the correct result at the topmost position of a ranked list. Thus, MRR can take a maximum value of 1 and a minimum value of 0. The bigger the MRR value is, the better the technique is.

**Mean Average Precision@K (MAP@K):** *Precision@K* calculates the precision at the occurrence of every single relevant item (e.g., API class, code segment) in the ranked list. *Average Precision@K (AP@K)*

---

<sup>7</sup><http://lucene.apache.org/>

averages the *precision@K* for all relevant items within Top-K results for a code search query [220, 276]. *Mean Average Precision@K* is the mean of *Average Precision@K* for all queries ( $Q$ ) from the dataset. MAP@K of a technique can be defined as follows:

$$AP@K = \frac{\sum_{k=1}^K P_k \times rel_k}{|RR|}$$

$$MAP@K = \frac{\sum_{q \in Q} AP@K(q)}{|Q|}$$

Here,  $rel_k$  denotes the relevance function of  $k^{th}$  result in the ranked list that returns either 1 (i.e., relevant) or 0 (i.e., non-relevant),  $P_k$  denotes the precision at  $k^{th}$  result, and  $K$  refers to number of top results considered.  $RR$  is the set of relevant results for a query, and  $Q$  is the set of all queries.

**Mean Recall@K (MR@K):** Recall@K refers to the percentage of gold set items (e.g., API, code segment) that are correctly recommended for a code search query in the Top-K results by a technique [249]. Mean Recall@K (MR@K) averages such measures for all queries ( $Q$ ) in the dataset. It can be defined as follows:

$$MR@K(Q) = \frac{1}{|Q|} \sum_{q \in Q} \frac{|result(q, K) \cap gold(q)|}{|gold(q)|}$$

Here,  $result(q, K)$  refers to Top-K recommended APIs by a technique, and  $gold(q)$  refers to goldset APIs for each query  $q \in Q$ . The bigger the MR@K value is, the better the recommendation technique is.

**Query Effectiveness (QE):** It refers to the rank of first relevant document in the results list retrieved by a query. The metric approximates a developer’s effort in locating the first item of interest. Thus, the lower the effectiveness measure is, the more effective the query is [98, 164, 191]. We use this measure to determine whether a given query is improved or not after its reformulation.

**Normalized Discounted Cumulative Gain (NDCG):** It determines the quality of ranking provided by a technique. With a *graded relevance scale* for results, the metric accumulates overall gain or usefulness from the top to the bottom of the list [112, 253]. It assumes that (1) highly relevant results are more useful when they appear earlier in the ranked list, and (2) highly relevant results are more useful than marginally relevant results. Thus, Discounted Cumulative Gain (DCG) of a ranked list returned by a query  $q$  can be calculated as follows:

$$DCG(q) = \sum_{k=1}^K \frac{grel_k}{\log_2(k+1)} \quad \text{where} \quad grel_k = 1 - \frac{goldRank(k, gold(q))}{|gold(q)|} \quad (7.7)$$

Here,  $grel_k$  refers to the graded relevance of the result at position  $k$ .  $goldRank(\cdot)$  returns the rank of the  $k^{th}$  result within the goldset items  $gold(q)$ . If  $k^{th}$  result is not found in the goldset,  $grel_k$  simply returns 0 as a special case. Thus,  $grel_k$  provides a graded relevance scale between 0 and 1 for each relevant result. Once  $DCG(q)$  is calculated, the normalized DCG can be calculated as follows:

$$NDCG(q) = \frac{DCG(q)}{IDCG(q)}, \quad NDGC(Q) = \frac{1}{|Q|} \sum_{q \in Q} NCDG(q) \quad (7.8)$$



Here  $IDCG(q)$  is the Ideal Discounted Cumulative Gain which is derived from the ranking of goldset items. Thus,  $NDCG(q)$  is the metric for one single query  $q$ , whereas  $NDCG(Q)$  averages the metric over all queries ( $\forall q \in Q$ ). We use NDCG in order to determine the quality of code search ranking from the traditional web/code search engines (Section 7.4.13).

### 7.4.3 Evaluation Scenarios

Our work in this article has two different aspects— (a) relevant API suggestion and (b) automatic query reformulation. We thus employ two different setups for evaluating our approach. In the first case, we investigate API suggestion performance of RACK, calibrate our adopted parameters, and compare with the state-of-the-art approaches on API suggestion [243, 274] (RQ<sub>4</sub>–RQ<sub>8</sub>). In the second case, we reformulate the initial NL queries from the dataset using our suggested API classes. Then we compare our reformulated queries not only with the baseline queries but also with the queries generated by the state-of-the-art approaches on query reformulation [168, 274] (RQ<sub>9</sub>–RQ<sub>10</sub>). We also investigate the potential of our queries in the context of contemporary web and code search practices (RQ<sub>11</sub>).

### 7.4.4 Statistical Significance Tests

In our comparison studies, we perform two statistical tests before claiming significance of one set of items over the other. In particular, we employ *Mann-Whitney Wilcoxon (MWW)* and *Wilcoxon Signed Rank (WSR)* for significance tests. We refer to them as MWW and WSR respectively in the remaining sections. MWW is a non-parametric test that (1) does not assume normality of the data and (2) is appropriate for small dataset [98]. We use this test for comparing any two arbitrary lists of items. WSR test is another non-parametric test that performs pair-wise comparison between two lists. In our experiment, WSR was used for significance test between performance measures (e.g., Hit@K) of RACK in API/code suggestion and that of an existing approach for the same  $K$  positions (i.e.,  $1 \leq K \leq 10$ ) (RQ<sub>8</sub>, RQ<sub>9</sub>, RQ<sub>10</sub>). We report *p-value* of each statistical test, and use 0.05 as the significance threshold. In addition to these significance tests, we also perform effect size test using *Cliff’s delta* to demonstrate the level of significance. For this work, we use three significance levels – *short* ( $0.147 \leq \Delta \leq 0.33$ ), *medium* ( $0.33 \leq \Delta \leq 0.474$ ) and *large* ( $\Delta \geq 0.474$ ) [216]. We use two R packages – `stats`, `effsize` – for conducting these statistical tests.

### 7.4.5 Matching of Suggested APIs with Goldset APIs

In order to determine performance of a technique, we apply *strict* matching between gold set APIs and the recommended APIs. That is, we consider two API classes matched if (1) they are categorically the same, and (2) they are superclass or subclass of each other. For example, if `OutputStream` is a gold set API and `FileOutputStream` is a recommended API, we consider them and their inverse as matched. If a base class is relevant for a programming task, the derived class is also likely to be relevant and thus, the recommendation

**Table 7.7:** Performance of RACK

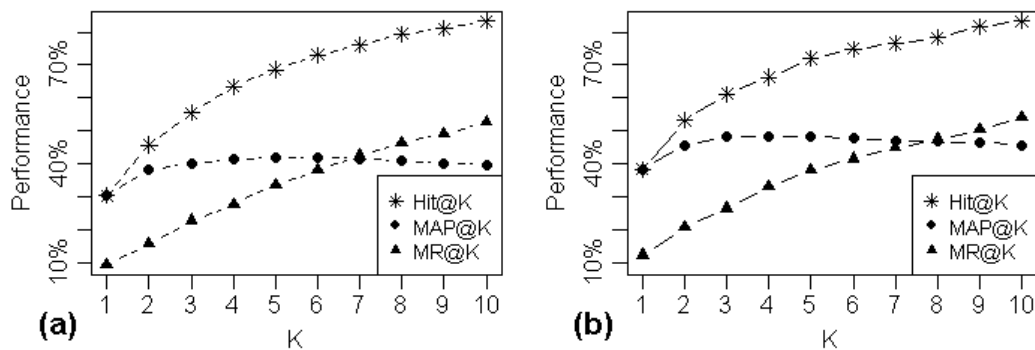
Metric	Non-weighted Version				Weighted Version			
	Top-1	Top-3	Top-5	Top-10	Top-1	Top-3	Top-5	Top-10
Hit@K	30.29%	55.43%	68.57%	<b>83.43%</b>	38.29%	61.14%	72.00%	<b>83.43%</b>
MRR@K	0.30	0.41	0.44	<b>0.46</b>	0.38	0.48	0.48	<b>0.52</b>
MAP@K	30.29%	40.19%	<b>42.00%</b>	39.66%	38.29%	48.14%	<b>48.39%</b>	45.74%
MR@K	9.24%	22.67%	33.53%	<b>52.78%</b>	12.12%	26.41%	37.94%	<b>54.07%</b>

is considered to be accurate. In the case of relevant code segment retrieval, we also apply exact matching between gold set segment and returned segment by a query. Since the tutorial sites clearly indicate which of the code segments implements which of our selected tasks (i.e., queries), such matching is warranted for this case. It should be noted that items (e.g., API class, code segment) outside the goldset could be also relevant to our queries. However, we stick to our gold sets for the sake of simplicity and clarity of our experiments. Our gold sets are also publicly available [201] for third-party replication or reuse.

#### 7.4.6 Answering RQ<sub>4</sub>: How does the proposed technique perform in suggesting relevant APIs for a code search query?

Each of our selected queries summarizes a programming task that requires the use of one or more API classes from various Java libraries. Our technique recommends Top-K (e.g.,  $K = 10$ ) relevant API classes for each query. We compare the recommended items with the *API-goldset* and evaluate them using above four metrics. In this section, we answer RQ<sub>4</sub> using Table 7.7 and Fig. 7.13.

Table 7.7 shows the performance details of our technique for Top-1, Top-3, Top-5 and Top-10 API recommendations. We see that our technique recommends at least one API correctly for 83%+ of the queries with both its (a) non-weighted and (b) weighted versions. The weighted version applies a fine tuned weight to each of our three heuristics—KAC, KPAC and KKC—whereas the non-weighted version treats each of the heuristics equally. Such accuracy is highly promising according to the relevant literature [63, 152]. Mean average precision and mean recall of RACK are 40%–46% and 53%–54% respectively for Top-10 results which are also promising. It also should be noted that RACK provides 55%–61% accuracy and 40%–48% precision for only Top-3 results which are good. That means, one out of the two suggested API classes is found to be relevant for the task, which could be really helpful for effective code search. Our mean reciprocal ranks are 0.46 and 0.52 for non-weighted and weighted version respectively. That is, the first correct suggestion is generally found between first to second position of our ranked list, which demonstrates the potential of our technique. Fig. 7.13 shows how different performance metrics – accuracy, precision and recall– change over different values of Top- $K$ . We see that our technique reaches a high precision (i.e., 48.14%) quite early (i.e.,  $K = 3$ ) and the highest (i.e., 48.39%) at  $K = 5$ , and then stays comparable for the rest of the  $K$  values. However, the improvement of recall measure is comparatively slow. It is  $\approx 10\%$  for  $K = 1$ , and then increases



**Figure 7.13:** Hit@K, Mean Average Precision@K, and Mean Recall@K of RACK using (a) non-weighted version (i.e., dashed line) and (b) weighted version (i.e., solid line)

somewhat linearly up to 54% for the last value of  $K = 10$ . On the contrary, the accuracy of RACK improves in a log-linear fashion, and becomes somewhat stationary for  $K = 10$  with 83%. While our accuracy and recall could further improve for increased  $K$ -values, the precision is likely to drop. Thus, we conduct our experiments using only Top-10 suggestions from a technique. Developers generally do not check items beyond the Top-10 items from the ranked list, and relevant literature [191, 239] also widely apply such cut-off value. Thus, our choice of  $K = 1$  to 10 is also justified.

We also analyse the distribution of API classes from 19 (11 core + 8 non-core) Java packages (i.e., Table 7.1) in our ground truth, and investigate how they correlate with corresponding distributions from Stack Overflow. We found that on average, 10% of the standard Java API classes from each package overlap with our ground truth classes. On the contrary, 65% of the API classes from each package are discussed in Stack Overflow Q & A threads according to RQ<sub>2</sub>. Thus, Stack Overflow discusses more API classes than the ground truth warrants for. In short, Stack Overflow is highly likely to deliver the relevant classes from standard API packages, and our approach harnesses that power. We also found that 51% of the ground truth classes come from the core packages whereas 10% of them come from the non-core packages. Since Stack Overflow has a good coverage (e.g.,  $\approx 65\%$ ) for both core and non-core packages (Fig. 7.7), RACK is also likely to perform well for such queries that require the API classes from non-core packages only.

We also determine correlation between four performance measures (e.g., Hit@10, reciprocal rank, average precision, recall) of our API suggestions (against NL queries) and the coverage of their corresponding ground truth in the constructed API database (Section 7.3.1). We employed two correlation methods – *Pearson* and *Spearman*, and found either very weak or negligible correlations (i.e.,  $0.04 \leq \rho \leq 0.12$ ) between those two entities. That is, the API suggestion performance of RACK is not biased by the coverage of the ground truth API classes in our API database. Such finding strengthens the external validity of our results.

**Summary of RQ<sub>4</sub>:** RACK suggests relevant API classes for about **83%** of the generic NL queries with a mean average precision@10 of **40%–46%**, a mean reciprocal rank@10 of **0.46–0.52**, and a mean recall@10 of **53%–54%**, which are highly promising.

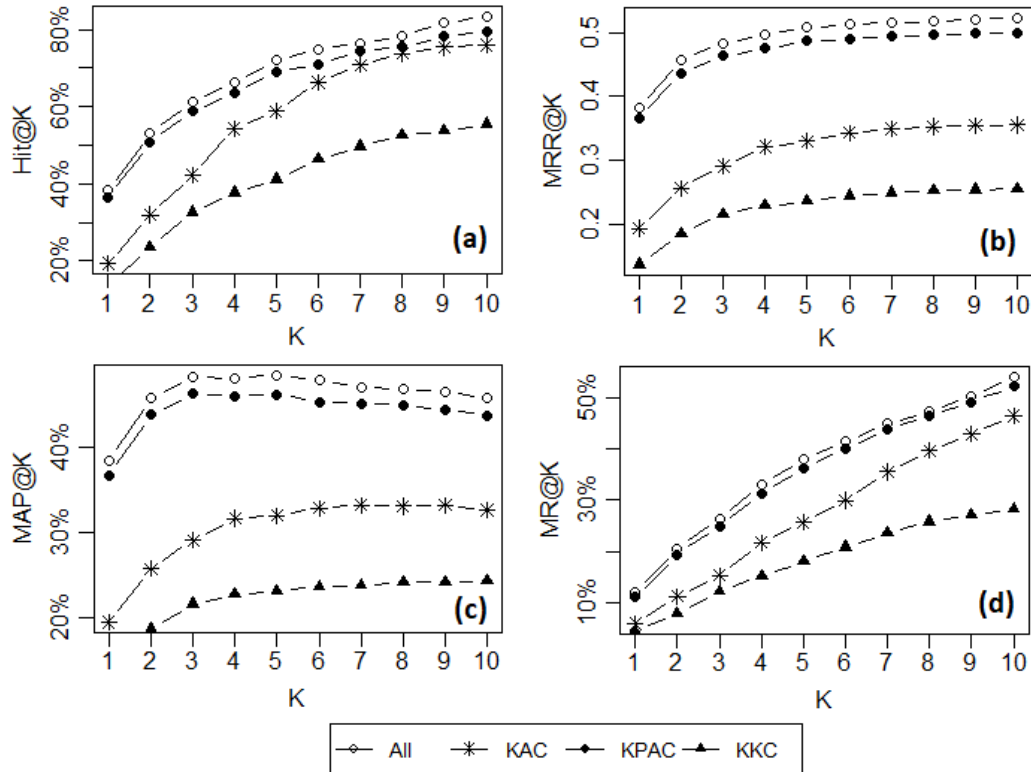
**Table 7.8:** Role of Proposed Heuristics– KAC, KPAC and KKC

Heuristics	Metric	Top-1	Top-3	Top-5	Top-10
{Keyword-API Co-occurrence (KAC)}	Hit@K	19.43%	42.29%	58.86%	76.00%
	MRR@K	0.19	0.29	0.33	0.36
	MAP@K	19.43%	29.05%	31.94%	32.57%
	MR@K	5.97%	15.35%	25.71%	46.42%
{Keyword Pair-API Co-occurrence (KPAC)}	Hit@K	36.57%	58.86%	69.14%	<b>79.43%</b>
	MRR@K	0.37	0.46	0.49	<b>0.50</b>
	MAP@K	36.57%	<b>46.19%</b>	46.13%	43.65%
	MR@K	11.08%	24.88%	36.20%	<b>52.21%</b>
{Keyword-Keyword Coherence (KKC)}	Hit@K	13.71%	32.57%	41.14%	55.43%
	MRR@K	0.14	.22	0.24	0.26
	MAP@K	13.71%	21.52%	23.05%	24.26%
	MR@K	4.46%	12.32%	18.07%	28.29%
{KAC + KKC} [201]	Hit@K	17.71%	40.00%	58.29%	<b>77.71%</b>
	MRR@K	0.18	0.28	0.32	0.34
	MAP@K	17.71%	27.57%	30.24%	30.84%
	MR@K	5.65%	14.66%	25.56%	<b>46.15%</b>
<b>RACK</b> [206]	Hit@K	38.29%	61.14%	72.00%	<b>83.43%</b>
	MRR@K	0.38	0.48	0.48	<b>0.52</b>
	MAP@K	38.29%	48.14%	<b>48.39%</b>	45.74%
	MR@K	12.12%	26.41%	37.94%	<b>54.07%</b>

#### 7.4.7 Answering RQ<sub>5</sub>: How effective are the proposed heuristics–KAC, KPAC and KKC– in capturing the relevant API classes for a query?

We investigate the effectiveness of our adopted heuristics– KAC, KPAC and KKC, and justify their combination in the API ranking algorithm (i.e., Algorithm 8). Table 7.8 and Fig. 7.14 demonstrate how each heuristic performs in capturing the relevant APIs for a given set of code search query as follows:

From Table 7.8, we see that our technique suggests correct API classes for 78.00% and 79% of the queries when KAC and KPAC heuristics are employed respectively. Both heuristics leverage co-occurrences between query keywords (in the question titles) and API classes (in the accepted answers) from Stack Overflow for such recommendation. On the contrary, KKC considers coherence among the candidate API classes, and is found less effective than the former two heuristics. In fact, KPAC performs the best among all three heuristics with up to 46% precision and 52% recall. However, the weighted combination of our heuristics provides the maximum performance in terms of four metrics. It provides 83% Hit@10 with a mean reciprocal rank@10 of 0.52, a mean average precision@10 of 46% and a mean recall@10 of 54%. That is, our combination harnesses



**Figure 7.14:** (a) Hit@K of RACK, (b) Mean Average Precision@K (MAP@K) of RACK, and (c) Mean Recall@K (MR@K) of RACK for three heuristics—KAC, KPAC and KKC

the strength from all the heuristics, and also overcomes their weaknesses simultaneously using appropriate weights. All these statistics are also highly promising according to the relevant literature [152, 243]. Thus, our combination of these three heuristics is also justified. Our earlier work combines KAC and KKC, and provides 79% Hit@10 with 35% precision and 45% recall from the experiments with 150 queries. Replication with our current extended dataset (i.e., 175 queries) reports similar performance (e.g., 78% Hit@10), which supports our earlier findings [201] as well. In this work, we introduce the new heuristic—KPAC—which improved our performance in terms of all four metrics—Hit@10 (i.e., 7% improvement), reciprocal rank (i.e., 53% improvement), precision (i.e., 48% improvement) and recall (i.e., 17% improvement). Thus, the addition of KPAC heuristic to our ranking algorithm is justified. Furthermore, we apply appropriate weights to these heuristics for controlling their influence in the API relevance ranking. Fig. 7.14 further demonstrates how the performance of our heuristics changes over various Top-K results. We see that KPAC is the most dominant one among the heuristics (as observed above) and achieves the maximum performance. However, the addition of the other two heuristics also improves our performance marginally (i.e., 2% – 4%) in terms of all four metrics.

**Summary of RQ<sub>5</sub>:** KPAC and KAC are found more effective than KKC in capturing the relevant API classes from Stack Overflow Q & A threads. However, combination of all three heuristics using appropriate relative weights delivers the maximum performance. Thus, their combination for API ranking is justified.

**Table 7.9:** Impact of Different Query Term Selection

Query Terms	Metric	Top-1	Top-3	Top-5	Top-10
All terms from query	Hit@K	37.14%	60.57%	71.43%	82.86%
	MRR@K	0.37	0.48	0.50	0.52
	MAP@K	37.14%	47.29%	47.81%	45.29%
	MR@K	11.69%	26.93%	38.85%	54.80%
Noun terms only	Hit@K	33.71%	58.86%	70.29%	82.29%
	MRR@K	0.34	0.45	0.48	0.49
	MAP@K	33.71%	44.95%	45.71%	42.62%
	MR@K	10.56%	25.47%	36.67%	55.21%
Verb terms only	Hit@K	7.43%	17.71%	24.00%	35.43%
	MRR@K	0.07	0.11	0.13	0.14
	MAP@K	7.43%	11.52%	12.68%	14.02%
	MR@K	2.14%	6.69%	10.46%	17.38%
{Noun terms + Verb terms}	Hit@K	38.29%	61.14%	72.00%	83.43%
	MRR@K	0.38	0.48	0.48	<b>0.52</b>
	MAP@K	38.29%	48.14%	48.39%	45.74%
	MR@K	12.12%	26.41%	37.94%	54.07%
{Noun terms + Verb terms}-"java"	Hit@K	37.14%	60.57%	72.00%	83.43%
	MRR@K	0.37	0.47	0.47	0.52
	MAP@K	37.14%	46.90%	47.35%	45.19%
	MR@K	11.84%	26.18%	38.09%	54.08%

#### 7.4.8 Answering RQ<sub>6</sub>: Does an appropriate subset of the query keywords perform better than the whole query in retrieving the relevant API classes?

Since the proposed technique identifies relevant API classes based on their co-occurrences with the keywords from a query, the keywords should be chosen carefully. Selection of random keywords might not return appropriate API classes. Several earlier studies choose nouns and verbs from a sentence, and report their salience in automated comment generation [259] and corpus indexing [59]. We thus also extract noun and verb terms from each query as the search keywords using Stanford POS tagger [244], and then use them for our experiments. In particular, we investigate whether our selection of keywords for code search is effective or not.

From Table 7.9, we see that our technique performs better with *noun*-based keywords than with *verb*-based keywords. The verb-based keywords provide a maximum of 35% Hit@10. On the contrary, RACK returns correct API classes for 82% of queries with 43% precision, 55% recall and a reciprocal rank of 0.49 when only noun-based keywords are chosen for search. However, none of the performance metrics reaches the

baseline performance except recall. That is, they are lower than the performance of RACK with all query terms minus the stop words. Interestingly, when both nouns and verbs are employed as search keywords, the performance reaches the maximum especially in terms of accuracy, precision and reciprocal rank. For example, RACK achieves 83% Hit@10 with 46% precision, 54% recall and a reciprocal rank of 0.52. Although the improvement over the baseline performance (i.e., with all keywords of a query) is marginal, such performances were delivered using a fewer number of search keywords. That is, our subset of keywords not only avoids the noise but also ensures a comparatively higher performance than the baseline with relatively lower costs (i.e., fewer keywords). Thus, selection of a subset of keywords from the NL query intended for code search is justified, and our subset is also found effective.

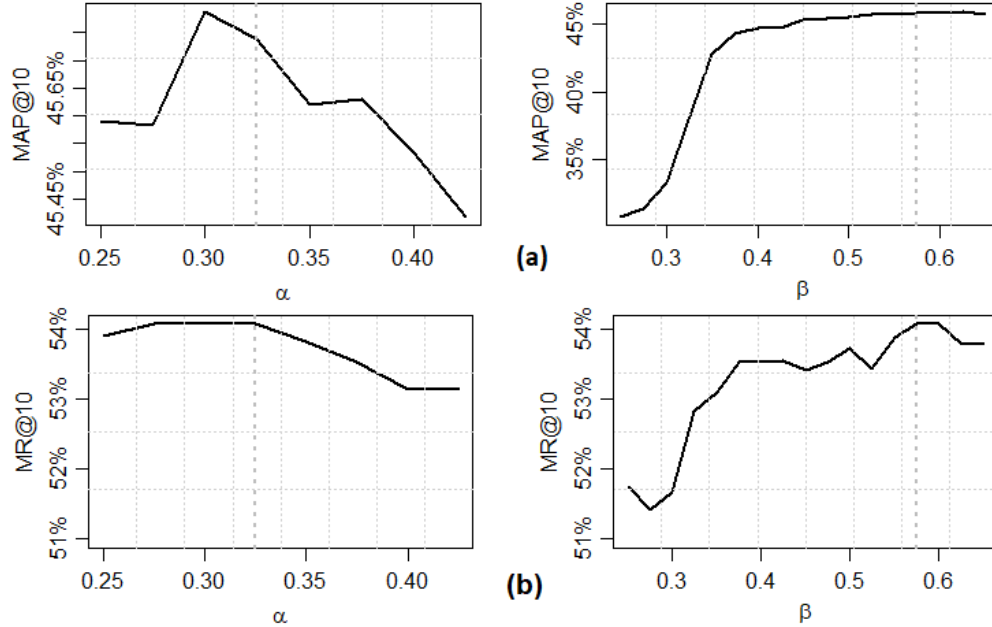
We also investigate the impact of generic search keywords such as “java” in our query. According to our analysis, 11.43% of our queries in the dataset contain this keyword. From Table 7.9, we see that removal of this keyword marginally degrades most of the performance measures of our technique. Only marginal improvements can be observed in the recall measure for Top-5 and Top-10 results. Thus, our choice of retaining the generic keywords is also justified.

**Summary of RQ<sub>6</sub>:** Important keywords from a natural language query mainly consist of its **noun** and **verb** terms. Our keyword selection approach of leveraging noun and verbs from a query is found quite effective in the relevant API suggestion.

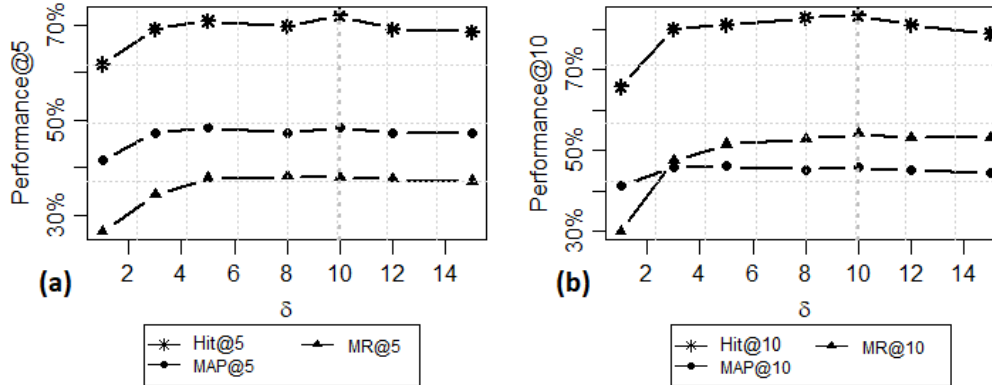
#### 7.4.9 Answering RQ<sub>7</sub>: How do the heuristic weights (i.e., $\alpha$ , $\beta$ ) and threshold settings (i.e., $\gamma$ , $\delta$ ) influence the performance of our technique?

Our relevance ranking algorithm applies two relative weights— $\alpha$  and  $\beta$ —to our proposed heuristics, and the heuristics are also constrained with two thresholds— $\gamma$  and  $\delta$ . While the thresholds help the heuristics collect appropriate candidate API classes, the weights control the influence of the heuristics in the API relevance ranking. In this section, we justify our chosen weights and thresholds, and investigate how they affect the performance of our technique.

We adopt a greedy search-based technique [272] (i.e., controlled iterative approach) for determining the relative weights for our heuristics. That is, we start our searches with our best initial guesses for  $\alpha$  (i.e., 0.25) and  $\beta$  (i.e., 0.30), refine our weight estimates in every iteration with a step size of 0.025, and then stop when the *fitness function* [272] (i.e., performance) reaches the global maximum. We use mean average precision@10 and mean recall@10 as the fitness functions in the search for  $\alpha$  and  $\beta$ . Fig. 7.15 shows how different values of  $\alpha$  and  $\beta$  can influence the performance of RACK. Please note that when one weight is calibrated, the other one is kept constant during performance computation. We see that precision and recall of RACK reach the maximum when  $\alpha \in [0.300, 0.325]$  and  $\beta=0.575$ . The target weights are identified using dashed vertical lines above. While  $\alpha$  and  $\beta$  are considered as the relative importance of the co-occurrence based heuristics, KAC and KPAC respectively,  $(1 - \alpha - \beta)$  goes to the remaining heuristic—KKC. Since KKC is found relatively



**Figure 7.15:** (a) Mean Average Precision@10 (MAP@10), and (b) Mean Recall@10 (MR@10) of RACK for different values of the heuristic weights— $\alpha$  and  $\beta$

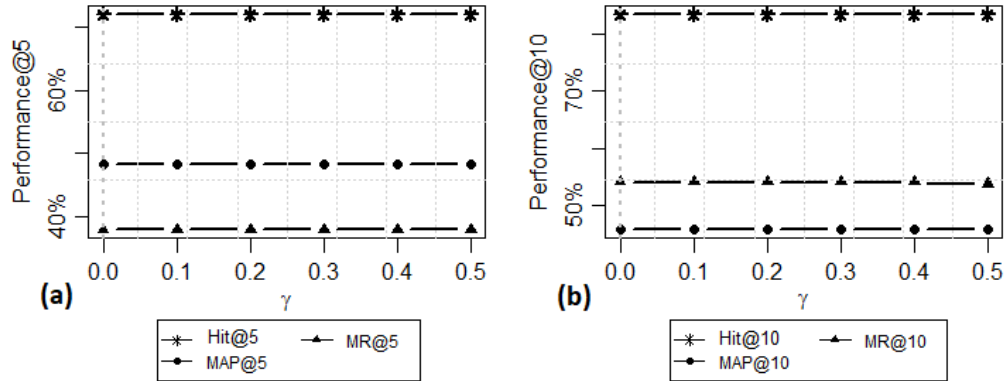


**Figure 7.16:** Performance of RACK for different  $\delta$  thresholds with (a) Top-5 results and (b) Top-10 results considered

weak according to our earlier investigation, we emphasize more on  $\alpha$  and  $\beta$ , and chose the following heuristic weights: 0.325, 0.575 and 0.10—for KAC, KPAC and KKC respectively. Thus, all the weights sum to 1, and such weighting mechanism was also used by an earlier study [163]. The performance of RACK is significantly higher than its non-weighted version especially in terms of MRR@K (i.e., WSR,  $p$ -value= 0.002,  $\Delta = 1.00$  (large)) and MAP@K (i.e., WSR,  $p$ -value< 0.001,  $\Delta = 0.84$  (large)) for Top-1 to Top-10 results. Thus, the application of relative weights to our adopted heuristics is also justified.

Both KAC and KPAC apply  $\delta$  threshold for collecting candidate API classes from the token-API linking database. Fig. 7.16 shows how different values of  $\delta$  can affect our performance. We use Hit@K, MAP@K and MR@K as the fitness functions, and determine our fitness for Top-5 and Top-10 returned results. We see that each of these performance measures reach their maximum when  $\delta = 10$  for both settings. That is, collecting





**Figure 7.17:** Performance of RACK for different  $\gamma$  thresholds with (a) Top-5 results and (b) Top-10 results considered

10 candidate API classes for each keyword or keyword pair from the query is the most appropriate choice. Less or more than that provides comparable performance but not the best one. Thus, we chose  $\delta = 10$  in our algorithm, and our choice is justified.

KKC applies another threshold,  $\gamma$ , for candidate API selection that refers to the degree of contextual similarity between any two keywords from the query. Fig. 7.17 reports our investigation on this threshold. We see that different values of  $\gamma$  starting from 0 to 0.5 do not change our fitness (i.e., performance) at all. Since the heuristic itself, KKC, is not strong, the variance of  $\gamma$  also does not have much influence on the performance of our technique. Thus, our choice of  $\gamma = 0$  is also justified. That is, we consider two API classes coherent to each other when their contexts share at least one search keyword.

**Summary of RQ<sub>7</sub>:** The performance of RACK reaches the *maximum* for certain weights and thresholds,  $\alpha=0.325$ ,  $\beta=0.575$ ,  $\gamma=0$ , and  $\delta=10$ . They were chosen carefully based on controlled iterative experiments, as were also done by the earlier studies [163, 272] from relevant literature.

#### 7.4.10 Answering RQ<sub>8</sub>: Can RACK outperform the state-of-the-art techniques in suggesting relevant API classes for a given set of queries?

Thung et al. [243] accept a feature request as an input and return a list of relevant API methods. Their API suggestions are based not only on the mining of feature request history but also on the textual similarity between the request texts and the corresponding API documentations. Zhang et al. [274] determine semantic distance between an NL query and a candidate API using a neural network model (CBOW) and a large code repository, and then suggest a list of relevant API classes for the query. To the best of our knowledge, these are the latest and the closest studies to ours in the context of API suggestion, and thus, we select them for comparison.

Since feature request history is not available in our experimental settings, we implement *Description-Based Recommender* module from Thung et al. We collect API documentations of 3,300 classes from the

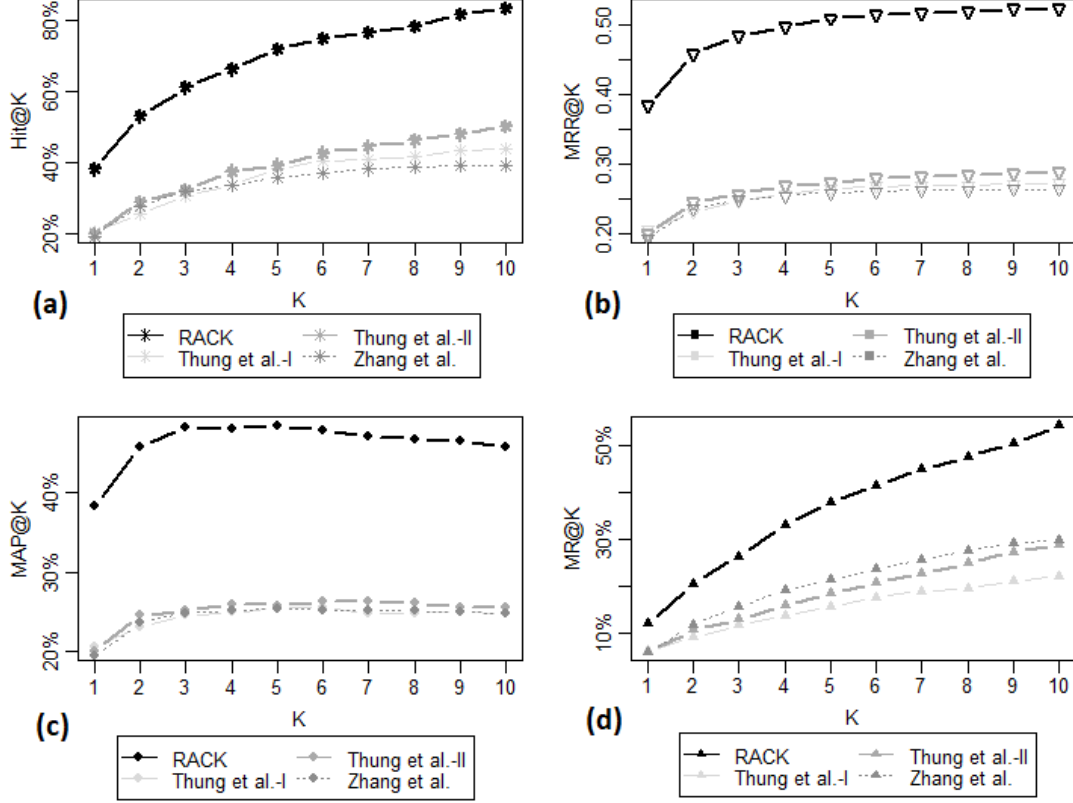
**Table 7.10:** Comparison of API Recommendation Performance with Existing Techniques (for various Top-K Results)

Technique	Metric	Top-1	Top-3	Top-5	Top-10
Thung et al. [243]-I	Hit@K	20.57%	30.85%	38.29%	44.00%
	MRR@K	0.21	0.25	0.26	0.27
	MAP@K	20.57%	24.57%	25.47%	24.84%
	MR@K	6.37%	11.74%	15.79%	22.19%
Thung et al. [243]-II	Hit@K	20.00%	32.57%	39.43%	<b>50.29%</b>
	MRR@K	0.20	0.26	0.27	<b>0.29</b>
	MAP@K	20.00%	25.14%	<b>25.85%</b>	25.59%
	MR@K	6.19%	13.02%	18.47%	28.95%
Zhang et al. [274]	Hit@K	19.43%	32.00%	36.00%	39.43%
	MRR@K	0.19	0.25	0.26	0.26
	MAP@K	19.43%	24.86%	25.44%	24.81%
	MR@K	6.00%	15.86%	21.54%	<b>29.87%</b>
<b>RACK</b> [206] (Proposed technique)	Hit@K	38.29%	61.14%	72.00%	<b>83.43%</b>
	MRR@K	0.38	0.48	0.48	<b>0.52</b>
	MAP@K	38.29%	48.14%	<b>48.39%</b>	45.74%
	MR@K	12.12%	26.41%	37.94%	<b>54.07%</b>

\***Emboldened** items are the highest statistics for the existing and proposed techniques

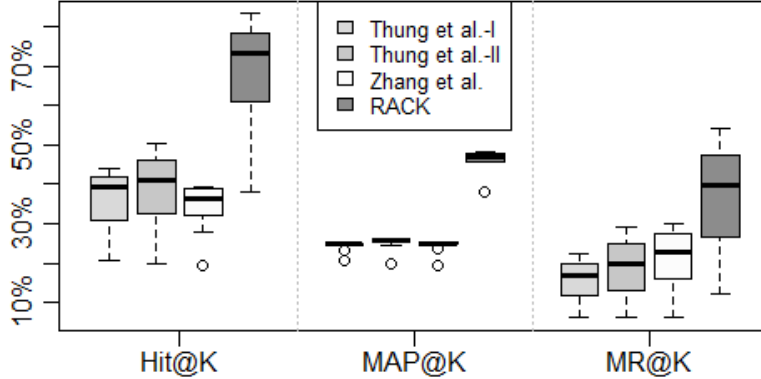
Java standard libraries (i.e., JDK 6), and develop Vector Space Model (VSM) for each of the API classes. In fact, we develop two models for each API class using (1) class header comments only, and (2) class header comments + method header comments, and implement two variants– Thung et al.-I and Thung et al.-II for our experiments. We use *Apache Lucene* [32] for VSM development, corpus indexing and for textual similarity matching between the API documentations and each of the queries from our dataset. In the case of Zhang et al., we (1) make use of *IJaDataset* [117] as a training corpus (as was done by the original authors), and (2) learn the word embeddings for both keywords and API classes using *fastText* [54], an improved version of *word2vec* implementation. We then use these vectors to determine semantic distance between a query and the candidate API classes using *cosine similarity* [198]. We also determine API popularity within the training corpus, and then combine with semantic distance metric to identify a set of relevant API classes for the NL query.

Table 7.10 summarizes the comparative analysis between our technique–RACK– and three existing techniques. Here, emboldened items refer to maximum measures provided by the existing techniques and our technique. We see that the variants of Thung et al. can provide a maximum of about 50% accuracy with about 26% precision and 29% recall for Top-10 results. On the other hand, RACK achieves a maximum



**Figure 7.18:** Comparison of API recommendation performances with the existing techniques-(a) Hit@K, (b) Mean Reciprocal Rank@K, (c) Mean Average Precision@K, and (d) Mean Recall@K

accuracy of 83% with 46% precision and 54% recall which are 66%, 79% and 87% higher respectively. We investigate how the four performance measures change for different Top-K results for each of these three techniques. From Fig. 7.18, we see that Hit@K of RACK increases gradually up to 83% whereas such performance measures for the textual similarity based techniques stop at 50%. The MRR@K of RACK improves from 0.38 to 0.52 whereas such measures for the counterparts are as low as 0.20–0.29. It should be noted that RACK reaches its maximum precision, i.e., 48%, quite early at  $K = 3$ , and then its recall gradually improves up to 54% (at  $K = 10$ ). On the contrary, such measures for the counterparts are at best 25% and 30% respectively. These demonstrate the superiority of our technique. From the box plots in Fig. 7.19, we see that RACK performs significantly higher than both variants in terms of all three metrics– accuracy, precision and recall. Our median accuracy is above 70% whereas such measures for those variants are close to 40%. The same goes for precision and recall measures. We perform significance and effect size tests, and compare our performance measures with the measures of the state-of-the-art for various Top-K results ( $1 \leq K \leq 10$ ). We found that the performance of our approach is significantly higher than that of the existing techniques in terms of Hit@K (i.e., WSR,  $p\text{-value}=0.002 < 0.05$ ,  $\Delta=0.79$  (large)), MRR@K (i.e., WSR,  $p\text{-value}=0.002 < 0.05$ ,  $\Delta=0.90$  (large)), MAP@K (i.e., WSR,  $p\text{-value}=0.002 < 0.05$ ,  $\Delta=0.90$  (large)) and MR@K (i.e., WSR,  $p\text{-value}=0.002 < 0.05$ ,  $\Delta=0.70$  (large)). All these findings above suggest that (1) textual similarity between query and API signature or documentations might not be always effective for API recommendation,



**Figure 7.19:** Comparison of API recommendation with existing techniques using box plots

and (2) semantic distance between keyword and API classes should be calculated using appropriate training corpus. Our technique overcomes that issue by applying three heuristics –KAC, KPAC and KKC– which leverage the API usage knowledge of a large developer crowd stored in Stack Overflow. Performance reported for Thung et al. is project-specific, and the technique is restricted to feature requests [243]. On the contrary, our technique is generic and adaptable for any type of code search. It is also independent of any subject systems. Although Zhang et al. employ a large training corpus, they learn word embeddings for NL keywords from the source code which might not be always helpful. Source code inherently has a smaller vocabulary than regular texts [102]. On the contrary, we leverage the contexts of NL keywords and API classes more carefully from Stack Overflow Q & A site to determine their relevance. Furthermore, we harnesses the expertise of a large crowd of technical users effectively for relevant API suggestion which was not considered by the past studies from literature. Thus, our technique possibly has a greater potential.

**Summary of RQ<sub>8</sub>:** Our approach, RACK, outperforms *multiple* existing studies on relevant API suggestion for NL queries, and achieves **66%** higher accuracy, **79%** higher precision and **87%** higher recall than those of the state-of-the-art.

#### 7.4.11 Answering RQ<sub>9</sub>: Can RACK significantly improve the natural language queries in terms of relevant code retrieval performance?

Our earlier research questions (RQ<sub>4</sub>–RQ<sub>8</sub>) evaluate the performance of RACK in suggesting relevant API classes for a natural language query intended for code search. Although they clearly demonstrate the potential of our technique, another way of evaluation could be the retrieval performance of our suggested queries. In this section, we investigate whether our reformulations to the baseline queries improve them or not in terms of their relevant code retrieval performances. We employ three corpora – *4K-Corpus*, *256K-Corpus*, and *769K-Corpus*– each of which includes 175 ground truth code segments (see Section 7.4.1 for details). We apply limited natural language preprocessing (i.e., removal of stop words and keywords, splitting of complex

**Table 7.11:** Comparison of Source Code Retrieval Performance with Baseline Queries

Query	Metric	Top-1	Top-3	Top-5	Top-10
<b>Retrieval Performance with Small Dataset (4K-Corpus)</b>					
Baseline	Hit@K	39.43%	54.86%	62.29%	68.57%
(NL Keywords)	MRR@K	0.39	0.46	0.48	0.49
Goldset API	Hit@K	65.71%	85.71%	89.14%	91.43%
	MRR@K	0.66	0.75	0.76	0.76
Baseline +	Hit@K	70.29%	88.00%	96.00%	97.14%
Goldset API	MRR@K	0.70	0.78	0.80	0.80
RACK <sub>A</sub>	Hit@K	29.71%	50.29%	56.00%	68.57%
	MRR@K	0.30	0.39	0.40	0.42
RACK <sub>A+Q</sub>	Hit@K	<b>50.86%</b>	<b>73.14%</b>	<b>77.71%</b>	<b>84.00%</b>
	MRR@K	<b>0.51</b>	<b>0.61</b>	<b>0.62</b>	<b>0.63</b>
<b>Retrieval Performance with Large Dataset (256K-Corpus)</b>					
Baseline	Hit@K	22.29%	30.86%	37.71%	44.00%
(NL Keywords)	MRR@K	0.22	0.26	0.27	0.28
Goldset API	Hit@K	60.00%	78.29%	84.57%	90.29%
	MRR@K	0.60	0.69	0.70	0.71
Baseline +	Hit@K	76.00%	89.14%	90.86%	94.86%
Goldset API	MRR@K	0.76	0.82	0.82	0.83
RACK <sub>A</sub>	Hit@K	14.29%	26.29%	30.86%	36.57%
	MRR@K	0.14	0.19	0.20	0.21
RACK <sub>A+Q</sub>	Hit@K	<b>40.00%</b>	<b>52.57%</b>	<b>59.43%</b>	<b>66.29%</b>
	MRR@K	<b>0.40</b>	<b>0.46</b>	<b>0.47</b>	<b>0.48</b>
<b>Retrieval Performance with Extra-Large Dataset (769K-Corpus)</b>					
Baseline	Hit@K	17.14%	24.57%	28.57%	34.29%
(NL Keywords)	MRR@K	0.17	0.20	0.21	0.22
Goldset API	Hit@K	50.86%	69.14%	75.43%	81.14%
	MRR@K	0.51	0.59	0.61	0.62
Baseline +	Hit@K	64.00%	80.00%	86.86%	90.29%
Goldset API	MRR@K	0.64	0.71	0.73	0.73
RACK <sub>A</sub>	Hit@K	10.86%	18.29%	22.29%	26.86%
	MRR@K	0.11	0.14	0.15	0.16
RACK <sub>A+Q</sub>	Hit@K	<b>26.86%</b>	<b>42.29%</b>	<b>49.14%</b>	<b>56.57%</b>
	MRR@K	<b>0.27</b>	<b>0.33</b>	<b>0.35</b>	<b>0.36</b>

**A**=Suggested API classes only, **A+Q**=Reformulated query combining both suggested API classes and baseline query keywords.

tokens) to each corpus document, and then index them for retrieval. We employ *Apache Lucene*<sup>8</sup>, a popular code search engine that has been used by several earlier studies from the literature [98, 163, 176], for document indexing and for source code retrieval.

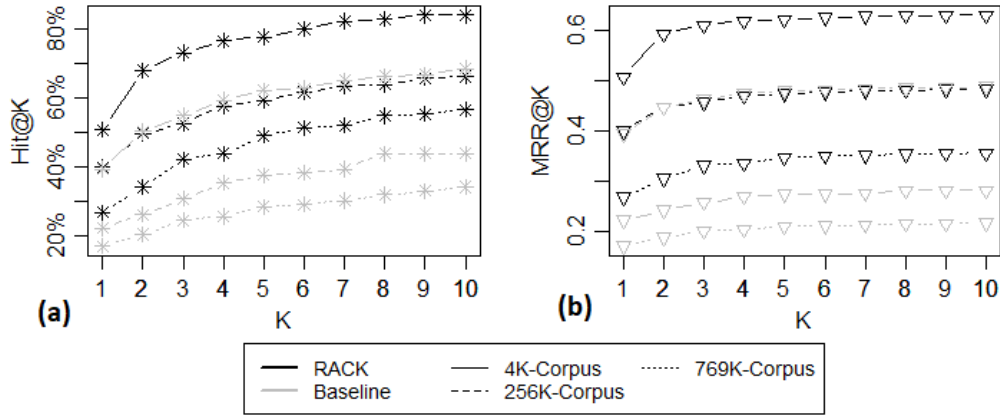
Table 7.11 and Fig. 7.20 summarize our findings on comparing our reformulated queries with the baseline queries. We consider two versions of our reformulated queries— $RACK_A$  and  $RACK_{A+Q}$ —for our experiments. While  $RACK_A$  comprises of suggested API classes only,  $RACK_{A+Q}$  combines both the suggested API classes and the NL keywords from baseline queries. From Table 7.11, we see that the baseline queries (i.e., comprise of NL keywords) perform poorly especially with the large corpora. In the case of *256K-Corpus*, they return relevant code segments at the Top-1 position and within the Top-5 positions for only 22% and 38% of the queries respectively (i.e., Hit@K). On the contrary, our reformulated queries,  $RACK_{A+Q}$ , can return relevant code segments for 40% and 59% of the queries within Top-1 and Top-5 positions respectively, which are more promising. We see a notable increase in the query performance with the smaller corpus (i.e., 4K-Corpus) and a notable decrease with the bigger corpus (i.e., 769K-Corpus). Such observations can be explained by the reduced and added noise in the corpus respectively. However, our reformulated queries perform consistently higher than the baseline across all three corpora. For example, while the baseline Hit@10 reduces to 34% for 769K-Corpus, our reformulated queries deliver a Hit@10 of 57% which is 65% higher. Thus, our query reformulations offer 23%-80% improvement in Hit@K over the baseline performance across the three corpora. It should be noted that Hit@1 and Hit@5 could reach up to 60% and 85% respectively when the goldset API classes are used as the search queries. Combination of NL queries and goldset API classes performs even better. Such findings also strengthen our idea of suggesting and using relevant API classes for code search. However, we also see that reformulated queries containing both NL keywords and API classes (e.g.,  $RACK_{A+Q}$ ) are always better than those containing only the suggested API classes (e.g.,  $RACK_A$ ).

Our MRR@K measures in Table 7.11 are also found more promising. They suggest that on average, the relevant code segments are returned by our queries within the top three positions of the result list across all three corpora, which is promising from the perspective of practical use. Furthermore, our MRR@K measures are 29%–81% higher than the baseline counterparts across all three corpora which demonstrate the potential of our reformulated queries for code search.

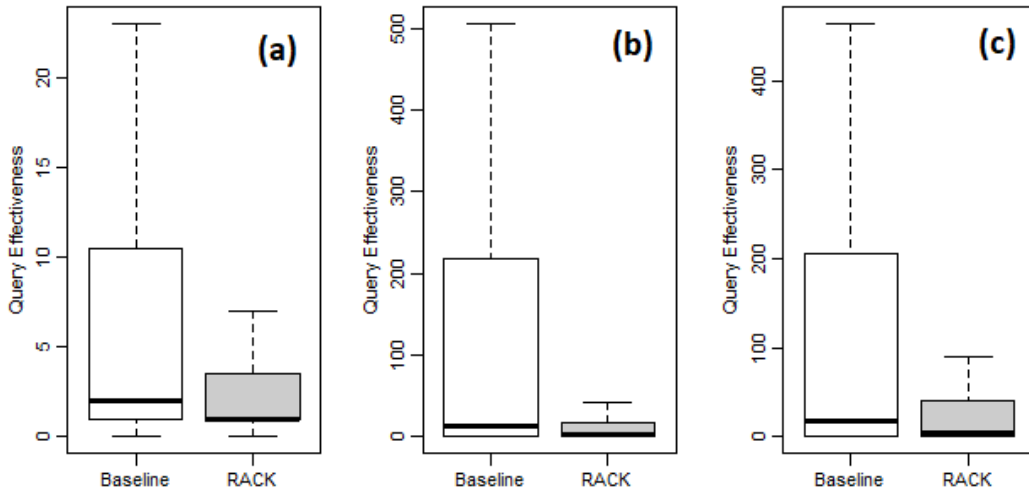
Fig. 7.20 further demonstrates the performance of baseline queries and our reformulated queries for various Top-K results. We see that Hit@K and MRR@K of our queries are higher than those of the baseline queries by a large margin across all three corpora—*4K-Corpus*, *256K-Corpus*, and *769K-Corpus*. Non-parametric tests such as *Wilcoxon Signed Rank*, *Mann-Whitney Wilcoxon* and *Cliff’s delta* tests also report statistical significance of our performance improvements for both Hit@K (i.e., all  $p\text{-values} < 0.05$ ,  $0.82 \leq \Delta \leq 0.94$  (large)) and MRR@K (i.e., all  $p\text{-values} < 0.05$ ,  $\Delta = 1.00$  (large)). For the sake of simplicity, only one code segment (i.e., collected from the tutorial sites, Section 7.4.1) was chosen as the ground truth of each query. Thus, Hit@K

---

<sup>8</sup><https://lucene.apache.org/>



**Figure 7.20:** Comparison of code retrieval performance with the baseline queries in terms of (a) Hit@K and (b) MRR@K



**Figure 7.21:** Comparison of QE distribution with baseline queries across (a) 4K-Corpus, (b) 256K-Corpus and (c) 769K-Corpus

and MRR@K are the most appropriate performance metrics for this case, and consequently, precision and recall were not chosen for this evaluation.

We also investigate query performance by relaxing the Top-K constraint and by analysing all the results returned by each query. Table 7.12 and Fig. 7.21 report our findings on query effectiveness [163, 164]. That is, if the first relevant code segment by a reformulated query is returned closer to the top of the result list than that of the baseline query, we consider it as *query quality improvement*, and vice versa as *query quality worsening*. If there is no change in the result ranks between baseline and reformulated queries, we call it *query quality preserving*. From Table 7.12, we see that 46%–64% of the baseline queries can be improved by our technique, RACK<sub>A+Q</sub>, across all three corpora. It worsens only 11%–16% of the queries, and thus, offers a net gain of 35%–49% query improvement. While 60% net gain is possible in the best case scenario using gold set APIs directly, our technique delivers  $\approx 50\%$ , which is promising according to relevant literature [98, 191]. Fig. 7.21 further contrast between baseline and our reformulated queries. We see that the result

**Table 7.12:** Improvement of Baseline Queries by RACK

Query Pairs	Improved	Worsened	Net Gain	Preserved
<b>Query Improvement with Small Dataset (4K-Corpus)</b>				
Goldset API vs. Baseline	<b>54.29%</b>	13.71%	+40.58%	32.00%
RACK <sub>A</sub> vs. Baseline	42.29%	39.43%	+2.86%	18.29%
RACK <sub>A+Q</sub> vs. Baseline	<b>46.29%</b>	10.86%	<b>+35.43%</b>	<b>42.86%</b>
<b>Query Improvement with Large Dataset (256K-Corpus)</b>				
Goldset API vs. Baseline	<b>70.86%</b>	14.29%	+56.00%	14.86%
RACK <sub>A</sub>	43.43%	48.00%	-4.57%	8.57%
RACK <sub>A+Q</sub>	<b>61.71%</b>	13.14%	<b>+48.57%</b>	<b>25.14%</b>
<b>Query Improvement with Extra-Large Dataset (769K-Corpus)</b>				
Goldset API vs. Baseline	<b>74.86%</b>	14.86%	<b>+60.00%</b>	10.29%
RACK <sub>A</sub>	44.00%	48.00%	-4.00%	8.00%
RACK <sub>A+Q</sub>	<b>64.00%</b>	16.00%	<b>+48.00%</b>	<b>20.00%</b>

**Net Gain** = Gained improvement of result ranks through query reformulations

ranks provided by RACK are closer to zero (i.e., top of the list) across all three corpora. Such finding provides more evidence on the high potential of our suggested queries.

**Summary of RQ<sub>9</sub>:** Reformulated queries by RACK retrieve relevant code segments with **23%–80%** higher accuracy and **29%–81%** higher reciprocal rank than those of the baseline queries. Furthermore, RACK improves **46%–64%** of the baseline queries, and they return the results closer to the top of the list.

#### 7.4.12 Answering RQ<sub>10</sub>: Can RACK outperform the state-of-the-art techniques in improving the natural language queries intended for code search?

Although our reformulations improve the baseline queries significantly, we further validate them against the queries generated by existing techniques including the state-of-the-art. The study of Zhang et al. [274] is a closely related work to ours. They suggest relevant API classes for natural language queries intended for code search by analysing semantic distance between query keywords and API classes. Thung et al. [243] is another related study in the context of relevant API suggestion which was originally targeted for feature location (i.e., project-specific code search). Recently, Nie et al. [168] reformulate a query for code search by collecting pseudo-relevance feedback from Stack Overflow, and then by applying Rocchio’s expansion [213] to the query. Their tool QECK suggests software-specific terms from programming questions and answers as query expansions. To the best of our knowledge, these are the most recent and the most closely related work to ours in the context of query reformulation for code search which make them the state-of-the-art. We thus compare our technique with these three existing techniques [168, 243, 274] in terms of Hit@K, MRR@K and Query Effectiveness (QE).



**Table 7.13:** Comparison of Code Retrieval Performance with Existing Techniques

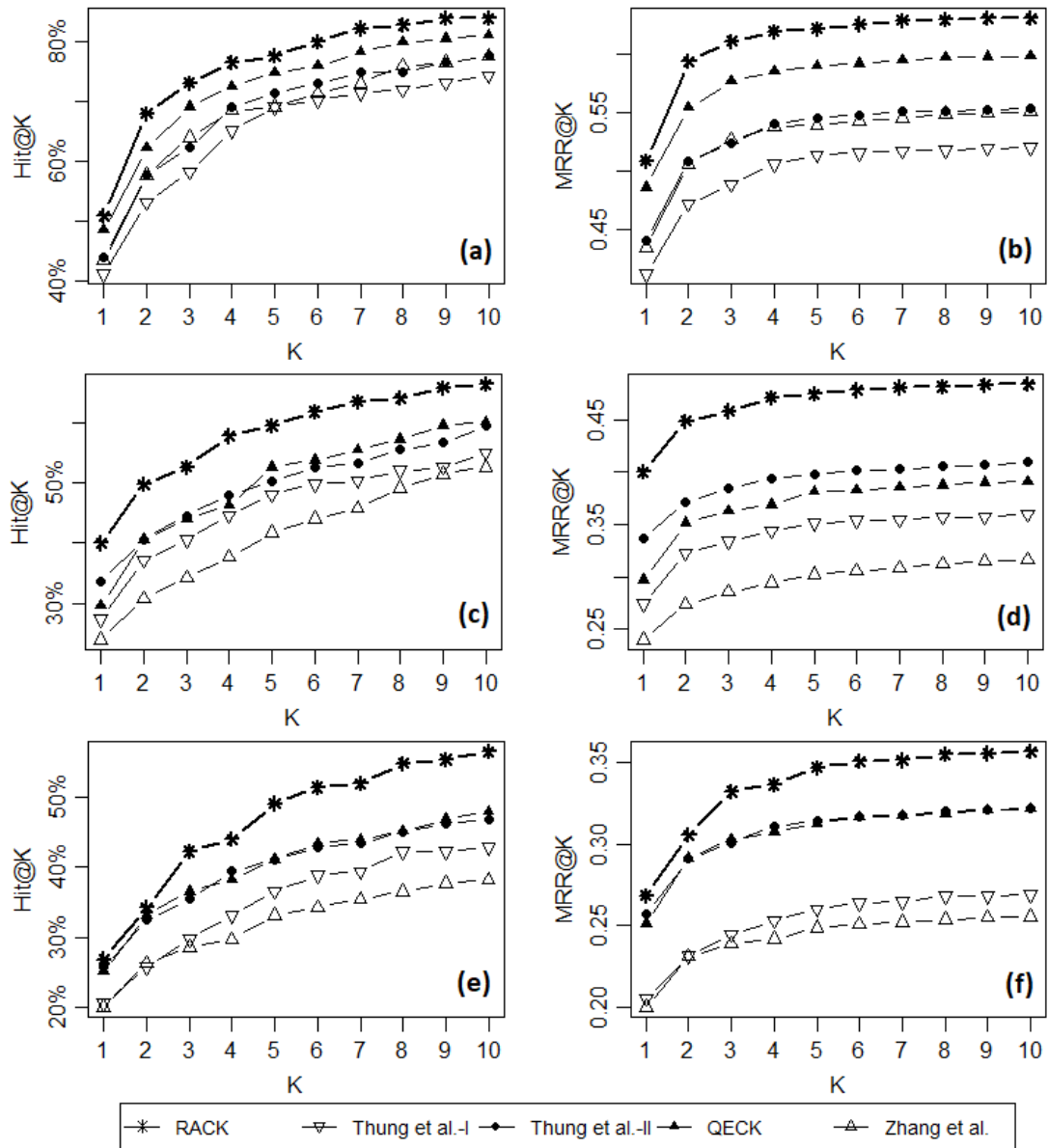
Technique	Metric	Top-1	Top-3	Top-5	Top-10
<b>Retrieval Performance with Small Dataset (4K-Corpus)</b>					
Thung et al. [243]-I	Hit@K	41.14%	58.29%	69.14%	74.29%
	MRR@K	0.41	0.49	0.51	0.52
Thung et al. [243]-II	Hit@K	44.00%	62.29%	71.43%	77.71%
	MRR@K	0.44	0.52	0.55	0.55
Nie et al. [168]	Hit@K	48.57%	69.14%	74.86%	<b>81.14%</b>
	MRR@K	0.49	0.58	0.59	<b>0.60</b>
Zhang et al. [274]	Hit@K	43.43%	64.00%	69.14%	77.71%
	MRR@K	0.43	0.53	0.54	0.55
<b>RACK</b> [206]	Hit@K	<b>50.86%</b>	<b>73.14%</b>	<b>77.71%</b>	<b>84.00%</b>
	MRR@K	<b>0.51</b>	<b>0.61</b>	<b>0.62</b>	<b>0.63</b>
<b>Retrieval Performance with Large Dataset (256K-Corpus)</b>					
Thung et al. [243]-I	Hit@K	27.43%	40.57%	48.00%	54.86%
	MRR@K	0.27	0.33	0.35	0.36
Thung et al. [243]-II	Hit@K	33.71%	44.57%	50.29%	59.43%
	MRR@K	0.34	0.39	0.40	0.41
Nie et al. [168]	Hit@K	29.71%	44.00%	52.57%	<b>60.00%</b>
	MRR@K	0.30	0.36	0.38	<b>0.39</b>
Zhang et al. [274]	Hit@K	24.00%	34.29%	41.71%	52.57%
	MRR@K	0.24	0.29	0.30	0.32
<b>RACK</b> [206]	Hit@K	<b>40.00%</b>	<b>52.57%</b>	<b>59.43%</b>	<b>66.29%</b>
	MRR@K	<b>0.40</b>	<b>0.46</b>	<b>0.47</b>	<b>0.48</b>
<b>Retrieval Performance with Extra-Large Dataset (769K-Corpus)</b>					
Thung et al. [243]-I	Hit@K	20.57%	29.71%	36.57%	42.86%
	MRR@K	0.21	0.24	0.26	0.27
Thung et al. [243]-II	Hit@K	25.71%	35.43%	41.14%	46.86%
	MRR@K	0.26	0.30	0.31	0.32
Nie et al. [168]	Hit@K	25.14%	36.57%	41.14%	<b>48.00%</b>
	MRR@K	0.25	0.30	0.31	<b>0.32</b>
Zhang et al. [274]	Hit@K	20.00%	28.57%	33.14%	38.29%
	MRR@K	0.20	0.24	0.25	0.26
<b>RACK</b> [206]	Hit@K	<b>26.86%</b>	<b>42.29%</b>	<b>49.14%</b>	<b>56.57%</b>
	MRR@K	<b>0.27</b>	<b>0.33</b>	<b>0.35</b>	<b>0.36</b>

From Table 7.13, we see that the retrieval performance of RACK is consistently higher than that of the state-of-the-art techniques or their variants across all three corpora. Nie et al. [168], performs the best among the existing techniques. Their approach achieves 41%–75% Hit@5 with a MRR@5 between 0.31 to 0.59 on our dataset. However, our technique, RACK, achieves 49%–78% Hit@5 with 0.35–0.62 MRR@5 which are 4%–19% and 5%–13% higher respectively. RACK also achieves a Hit@10 of 57% with the extra-large corpus (i.e., 769K-Corpus) which is 18% higher than the state-of-the-art measure, i.e., 48% Hit@10 by Nie et al. While the performance measures of each technique degrade as the corpus size grows from 4K to 769K documents, our performance measures remain consistently higher than the state-of-the-art. Thus, RACK is more robust to varying sizes of corpora than any of the existing techniques under our study.

Fig. 7.22 further demonstrates how RACK outperforms the state-of-the-art techniques for various Top-K results in terms of Hit@K and MRR@K. We compare RACK with QECK by Nie et al. [168] for Top-1 to Top-10 performance measures using non-parametric tests. Nie et al. is clearly the state-of-the-art according to the above analysis. Our *Mann-Whitney Wilcoxon* and *Cliff’s delta* tests reported statistical significance of RACK over Nie et al. with large effect sizes for both Hit@K (i.e.,  $p\text{-values} < 0.05$ ,  $0.33 \leq \Delta \leq 0.52$  (large)) and MRR@K (i.e.,  $p\text{-values} < 0.05$ ,  $0.68 \leq \Delta \leq 0.90$  (large)) across all three corpora. Thus, the findings above clearly demonstrate the superiority of our technique over the existing studies on query reformulation.

We also compare our technique with the existing techniques in terms of Query Effectiveness (QE). From Table 7.14, we see that Nie et al. performs the best with 4K-Corpus whereas Thung et al.-II performs the best with the remaining two corpora– 256K-Corpus and 769K-Corpus. Nie et al. improves 32% of the baseline queries whereas Thung et al.-II improves 43%–49% of the queries. On the contrary, RACK improves 46% and 62%–64% of the baseline queries in the same contexts. In particular, our technique offers 48% net gain as opposed to 26% provided by Thung et al.-II which is 87% higher. Thus, RACK clearly has a high potential for query reformulation than the state-of-the-art. It also should be noted that RACK degrades only 11%–16% of the queries across all three corpora which suggests the reliability and robustness of the technique. Fig. 7.23 further contrasts the result ranks of RACK with that of the state-of-the-art approaches using box plots. We see that on average, RACK provides higher ranks, and returns results closer to the top of list than the competing approaches. For example, Thung et al.-II returns 50% of its first correct results within the Top-8 positions and 75% of them within the Top-96 positions when dealing with extra-large corpus (i.e., 769K-Corpus). On the contrary, RACK returns such results within Top-5 and Top-42 positions which are 38% and 57% higher respectively. Similar findings can be observed with the remaining two corpora. All these findings above clearly demonstrate of superiority of our technique in query reformulation over the state-of-the-art.

**Summary of RQ<sub>10</sub>:** Reformulated queries of RACK retrieve relevant code segments with **19%** higher accuracy and **13%** higher reciprocal rank than the state-of-the-art. Furthermore, RACK offers **48%** net improvement in the quality of baseline queries, which is **87%** higher than the state-of-the-art counterpart.

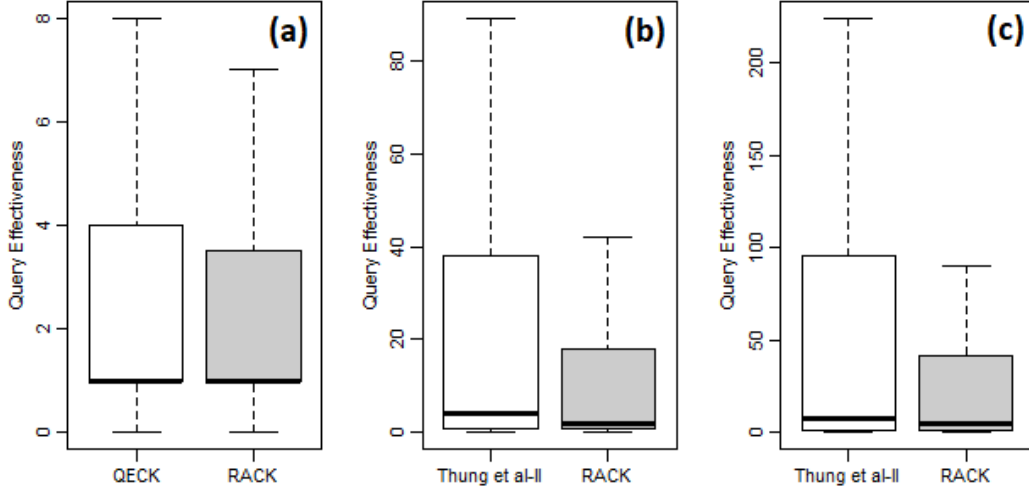


**Figure 7.22:** Comparison of code retrieval performance with existing techniques using (a,b) 4K-Corpus, (c,d) 256K-Corpus and (e,f) 756K-Corpus

**Table 7.14:** Comparison of Query Improvements with Existing Techniques

Query Pairs	Improved	Worsened	Net Gain	Preserved
<b>Query Improvement with Small Dataset (4K-Corpus)</b>				
Thung et al. [243]-I vs. Baseline	24.00%	11.43%	+12.57%	64.57%
Thung et al. [243]-II vs. Baseline	31.43%	10.86%	+20.57%	57.71%
Nie et al. [168] vs. Baseline	<b>32.00%</b>	8.00%	+ <b>24.00%</b>	60.00%
Zhang et al. [274] vs. Baseline	28.00%	10.29%	+17.71%	61.71%
<b>RACK vs. Baseline</b>	<b>46.29%</b>	10.86%	+ <b>35.43%</b>	<b>42.86%</b>
<b>Query Improvement with Large Dataset (256K-Corpus)</b>				
Thung et al. [243]-I vs. Baseline	37.71%	22.29%	+15.42%	40.00%
Thung et al. [243]-II vs. Baseline	<b>42.86%</b>	21.14%	+ <b>21.72%</b>	36.00%
Nie et al. [168] vs. Baseline	41.71%	24.57%	+17.14%	33.71%
Zhang et al. [274] vs. Baseline	36.00%	26.86%	+9.14%	37.14%
<b>RACK vs. Baseline</b>	<b>61.71%</b>	13.14%	+ <b>48.57%</b>	<b>25.14%</b>
<b>Query Improvement with Extra-Large Dataset (769K-Corpus)</b>				
Thung et al. [243]-I vs. Baseline	41.14%	25.71%	+15.43%	33.14%
Thung et al. [243]-II vs. Baseline	<b>48.57%</b>	22.86%	+ <b>25.71%</b>	28.57%
Nie et al. [168] vs. Baseline	45.71%	24.57%	+21.14%	29.71%
Zhang et al. [274] vs. Baseline	41.14%	28.57%	12.57%	30.29%
<b>RACK vs. Baseline</b>	<b>64.00%</b>	16.00%	+ <b>48.00%</b>	<b>20.00%</b>

**Net Gain** = Gained improvement of result ranks through query reformulations



**Figure 7.23:** Comparison of QE distribution with the state-of-the-art using (a) 4K-Corpus, (b) 256K-Corpus, and (c) 769K-Corpus

#### 7.4.13 Answering RQ<sub>11</sub>: How does RACK perform compared to the popular web search engines and code search engines?

Existing studies [166, 205, 219, 263] report that software developers frequently use general-purpose web search engines (e.g., Google) for code search. Hence, these search engines are natural candidates for comparison with our technique. We thus compare our approach with three popular web and code search engines— *Google*, *Stack Overflow native search* and *GitHub code search*. Unfortunately, we faced several challenges during our comparison with these commercial search engines. First, results from these search engines frequently change due to their dynamic indexing. This makes it hard to develop a reliable or stable oracle from their results. In fact, we found that Top-30 Google results collected for the same query in two different dates (i.e., two weeks apart) matched only 55%. Second, Google search API [8] was used for our experiments given that GUI based Google search is not a practical idea for  $175 \times 2 = 350$  queries. However, this paid search API imposes certain restrictions on the number of API calls to be made. That is, results for 175 baseline queries and their reformulated queries could not be collected all at the same time. Given the changing nature of the underlying corpus, comparison between the results of baseline and reformulated queries could thus not be fair. Third, these commercial search engines are mostly designed for natural language queries. They also impose certain restrictions on the query length and query type. Hence, they might either produce poor results or totally fail to produce any results for our reformulated queries which mostly contain structured keywords (e.g., multiple API classes). Thus, we found a head-to-head comparison with these commercial search engines infeasible. Despite the above challenges, we still compare with these engines, and investigate whether our reformulated queries can improve their search results significantly or not through a post-processing step of their results.

**Table 7.15:** Comparison with Popular Web/Code Search Engines

Technique	Hit@10	MAP@10	MRR@10	NDCG@10
Google	100.00%	68.56%	0.82	0.46
<b>RACK<sub>Google</sub></b>	100.00%	<b>83.71%</b>	<b>0.92</b>	<b>0.67</b>
Stack Overflow	91.43%	59.54%	0.67	0.43
<b>RACK<sub>SO</sub></b>	91.43%	<b>75.27%</b>	<b>0.82</b>	<b>0.62</b>
GitHub	89.71%	55.27%	0.58	0.47
<b>RACK<sub>GitHub</sub></b>	<b>90.29%</b>	<b>68.59%</b>	<b>0.74</b>	<b>0.59</b>

**Emboldened**= Comparatively higher than counterpart

**Collection of Search Results and Construction of Oracle:** We collect Top-30 results for each query from each search engine for oracle construction. We make use of *Custom Search API*<sup>9</sup> by Google and *native API endpoints* by Stack Overflow<sup>10</sup> and GitHub<sup>11</sup>, and collect the search results. Given the large volume of search results (i.e., 175 x 30 = 5,250), it is impractical to manually analyze them all. Hence, we used a semi-automated approach in constructing the oracle for these web/code search engines. In particular, we extract the code segments from each of the result pages using appropriate tools (e.g., Jsoup<sup>12</sup>). In the case of GitHub search results, we use *JavaParser*<sup>13</sup> to extract the method bodies as code segments. Then we determine their similarity against the original ground truth code that was extracted from tutorial sites in Section 7.4.1. For this, we use four code similarity algorithms – *Cosine similarity* [198], *Dice similarity* [98], *Jaccard similarity* [235] and *Longest Common Subsequence (LCS)* [218]. These algorithms are frequently used as the baseline for various code clone detection techniques [218, 235]. We collect four normalized code similarity scores from each result, average them, and then extract the Top-10 results containing the most relevant code segments. We then manually analyse a few of these results (and their code segments), and attempt to tweak them with various similarity score thresholds. Unfortunately, score thresholds were not sufficient enough to construct oracle for all the queries. We thus use these Top-10 results as the oracle for our web/code search engines.

**Comparison between Initial Search Results and Re-ranked Results using the Reformulated Queries:** Once a search engine returns results for natural language (NL) queries, we re-rank them with the corresponding reformulated queries provided by RACK. We first detect the presence of code segments in their contents, and then collect Top-10 documents based on their relevance to our reformulated queries (i.e., NL keywords + relevant API classes). We compare both the initial and re-ranked results with the oracle constructed above.

From Table 7.15, we see that our re-ranking approach improves upon the initial results returned by each of the web and code search engines. The improvements are observed especially in terms of precision,

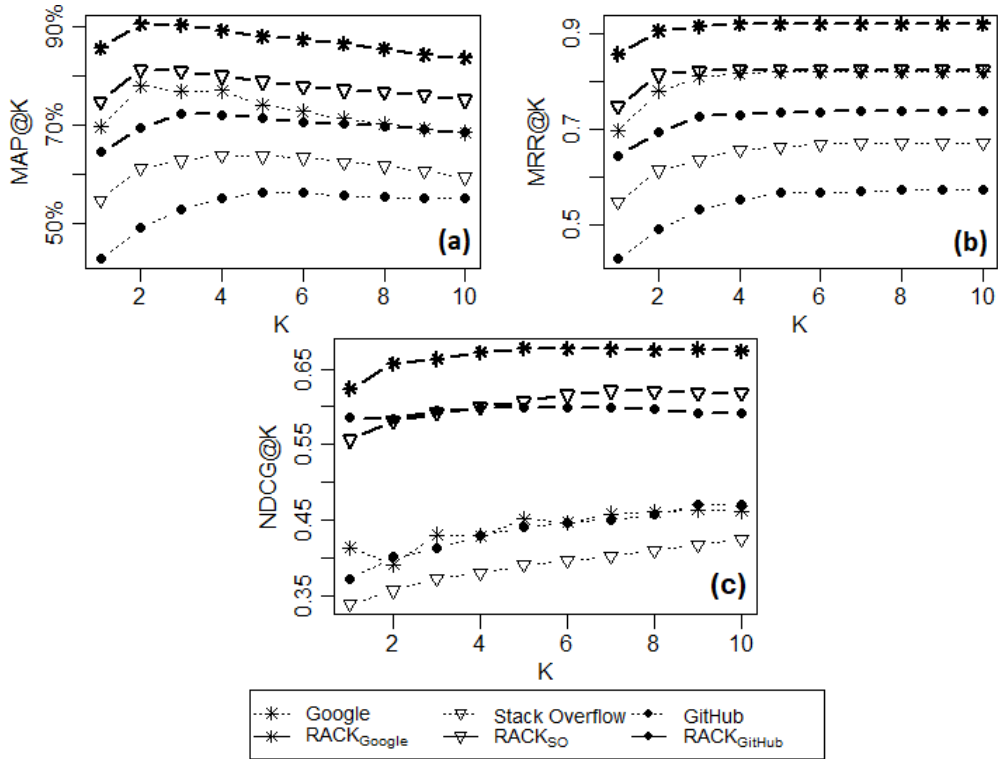
<sup>9</sup><https://developers.google.com/custom-search>

<sup>10</sup><https://api.stackexchange.com>

<sup>11</sup><https://developer.github.com/v3>

<sup>12</sup><https://jsoup.org/>

<sup>13</sup><https://github.com/javaparser>



**Figure 7.24:** Comparison of RACK with popular web/code search engines

reciprocal rank and NDCG. For example, Google achieves 69% precision with a reciprocal rank of 0.82 and an NDCG of 0.46. However, our approach,  $RACK_{Google}$  achieves 84% precision with a reciprocal rank of 0.92 and an NCCG of 0.67, which are 22%, 12% and 46% higher respectively. That is, although Google performs high as a *general-purpose* web search engine, it might not be always precise for code search. Similar observation is shared by a recent survey [205] that reports that developers need more query reformulations during code search using the web search engines. GitHub native search achieves 55% precision, a reciprocal rank of 0.58 and an NDCG of 0.47. On the contrary, our approach,  $RACK_{GitHub}$ , delivers 69% precision with a reciprocal rank of 0.74 and an NDCG of 0.62, which are 24%, 28% and 26% higher respectively. Such findings demonstrate the potential of our reformulated queries. Fig. 7.24 further contrasts between our approach and the contemporary web/code search engines for Top-1 to Top-10 results. While Google is the best performer among the three search engines, our re-ranking using RACK outperforms Google with a significant margin in terms of precision (i.e., WSR,  $p\text{-value} < 0.05$ ,  $\Delta = 0.90$  (large)), reciprocal rank (i.e., WSR,  $p\text{-value} < 0.05$ ,  $\Delta = 0.90$  (large)) and NDCG (i.e., WSR,  $p\text{-value} < 0.05$ ,  $\Delta = 1.00$  (large)). Thus, all the findings above suggest the high potential of our reformulated queries for improving the code search performed either with web or code search engines. The status quo of Internet-scale code search is far from ideal [205], and our reformulated queries could benefit the traditional practices.

**Table 7.16:** Comparison among the Traditional Code Search Engines

Serial	Question	Krugle <sup>a</sup>	SearchCode <sup>b</sup>	GitHub <sup>c</sup>	Codase <sup>d</sup>	Snipplr <sup>e</sup>
Q1	How to send email in Java?	N/A	1/5	1/5	N/A	0
Q2	How to calculate MD5 hash for a string?	N/A	4/5	5/5	N/A	0
Q3	How to parse HTML in Java?	N/A	0/5	1/5	N/A	1/5
Q4	How to parse XML in Java?	N/A	5/5	3/5	N/A	1/5
Q5	How to download a file in Java?	N/A	1/5	1/5	N/A	1/5

<sup>a</sup><https://www.krugle.com>, <sup>b</sup><https://searchcode.com>, <sup>c</sup><https://github.com/search>, <sup>d</sup><http://www.codase.com>, <sup>e</sup><https://snipplr.com/search.php>

One might also argue about our choice of *GitHub code search* over the other available code search engines (e.g., SearchCode, Krugle) for the comparison above. However, GitHub<sup>14</sup> has been the largest online codebase with 100 million repositories and 37 million users including the professional developers from the large companies (e.g., Microsoft, Google). Thus, GitHub is likely to contain a large collection of high quality code examples that implement numerous programming tasks, which could be leveraged by our approach. Despite this strong motivation, we conduct an experiment using five sample queries and five traditional code search engines (Table 7.16). In particular, we select the Top-5 code search engines according to a third-party survey<sup>15</sup>, execute the sample queries against them, and then report our findings. From Table 7.16, we see that Krugle and Codase accept the queries but fail to return any code segments. They might not be well calibrated for the free-form search queries. SearchCode was able to retrieve at least one somewhat relevant code example within the Top-5 results for 4 out of 5 queries. Interestingly, GitHub was able to retrieve at least one relevant code example for all five queries. Further analysis suggests that GitHub has its own codebase whereas SearchCode is dependent on GitHub for the results. GitHub also has a well managed API service which was crucial to our extensive experiment with 250 queries. We needed to make hundreds of API calls to collect the results against the search queries. In short, GitHub has the largest codebase online, a native search engine with public API service, and it performs better than the other traditional code search engines. Thus, our choice of using GitHub for the comparison study above is likely to be justified.

**Summary of RQ<sub>11</sub>:** Developers face difficulties in the code search while using contemporary web or code search engines (e.g., Google). Our technique can *significantly* improve their result ranks with the help of our reformulated queries that contain relevant API classes. In particular, RACK can improve upon the *precision* of Google in the code search by **22%**, which is promising. Our choice about web/code search engines for the comparison using experiments is also likely to be justified.

<sup>14</sup><https://en.wikipedia.org/wiki/GitHub>

<sup>15</sup><https://wparena.com/top-source-code-search-engines/>



## 7.5 Threats to Validity

We identify a few threats to the validity of our findings. While we attempt to mitigate most of them using appropriate measures, the remaining ones should be addressed in future work. Our identified threats and their mitigation details are discussed as follows:

### 7.5.1 Threats to Internal Validity

They relate to experimental errors and biases [272]. We develop a *gold set* for each query by analysing the code examples and the discussions from tutorial sites which might involve some subjectivity. However, each of the examples is a working solution to the corresponding task (i.e., NL-query), and they are frequently consulted. Thus, the gold set development using sample code from the tutorial sites is probably a more objective evaluation approach than human judgements of API relevance or code relevance that introduce more subjective bias [63]. According to the exploratory findings (Section 7.2.4), our technique might be effective only for the recommendation of popular and frequently used API classes. Since fully qualified names are mostly missing in Stack Overflow texts, third-party APIs similar to Java API classes could also have been mistakenly considered despite the fact that questions and answers selected for the study were tagged with `<java>` tag.

We use a dataset of 175 queries and a popular code search engine—*Apache Lucene* [98]—for determining their retrieval performance across three corpora of varying sizes. For the sake of simplicity, only one code segment was considered as relevant for each query. However, in practice, there could be multiple code segments in the corpus that are relevant to a given query. In this work, we trade such perfection with transparency and objectivity in our evaluation and validation.

During code or web search, developers generally choose the most appropriate keywords when a list of auto-generated suggestions are provided. We re-enact such behaviour of the developers by choosing only goldset API classes from within the suggested list, and use them for query reformulation. Such choice might have favoured the code retrieval performance of our queries. However, the same approach was carefully followed for all the existing techniques under study [168, 243, 274]. Thus, they received the same treatment in the performance evaluation as ours. Furthermore, the validation results (i.e., RQ<sub>10</sub>) clearly report the superiority of our suggested queries over their counterparts from the existing techniques. Our investigation using the three contemporary web/code search engines also has drawn a similar conclusion for RACK (i.e., RQ<sub>11</sub>).

### 7.5.2 Threats to External Validity

They relate to the generalizability of a technique. So far, we experimented using API classes from only standard Java libraries. However, since our technique mainly exploits co-occurrence between keywords and APIs, the technique can be easily adapted for API recommendation in other programming domains. Since

popularity of a programming language or change proneness of an API [140] has a significant role in triggering discussions at Stack Overflow which are mined by us, RACK could be effective for popular languages (e.g., Java, C#) but comparatively less effective for non-popular or less used languages (e.g., Erlang).

### 7.5.3 Threats to Construct Validity

Construct validity relates to suitability of evaluation metrics. Our work is aligned to both recommendation system and information retrieval domains. We use Hit@K and Reciprocal Rank which are widely used for evaluating recommendation systems [239, 243]. The remaining two metrics are well known in information retrieval, and are also frequently used by studies [63, 152, 243] relevant to our work. This confirms no or little threat to construct validity.

### 7.5.4 Threats to Statistical Conclusion Validity

Conclusion validity concerns the relationship between treatment and outcome [140]. We answer 11 research questions in this work, and collect our data from publicly available, popular programming Q & A and tutorial sites. In order to answer these questions, we use non-parametric tests for statistical significance (e.g., Mann-Whitney Wilcoxon, Wilcoxon Signed Rank), effect size analysis (e.g., Cliff’s delta) and confidence interval analysis. We apply these tests to our experiments opportunistically and report the detailed test results (e.g., p-values, Cliff’s delta). Thus, threats to the statistical conclusion validity might be mitigated.

## 7.6 Related Work

Our work is aligned with three research topics—(1) API/API usage recommendation, (2) query reformulation for code search, and (3) crowdsourced knowledge mining. In this section, we discuss existing studies from the literature of each of these research topics, and compare or contrast our work with them.

### 7.6.1 API Recommendation

Existing studies on API recommendation accept one or more natural language queries, and recommend relevant API classes and methods by analysing code surfing behaviour of the developers and API invocation chains [152], API dependency graphs [63], feature request history or API documentations [243], and library usage patterns [242]. McMillan et al. [152] first propose *Portfolio* that recommends relevant API methods for a code search query by employing natural language processing, indexing and graph-based algorithms (e.g., PageRank [57]). Chan et al. [63] improve upon *Portfolio*, and return a connected sub-graph containing the most relevant APIs by employing further sophisticated graph-mining and textual similarity techniques. Gvero and Kuncak [92] accept a free-form NL-query, and return a list of relevant method signatures by employing natural language processing and statistical language modelling on the source code. A few studies offer NL interfaces for searching relevant program elements from the project source [124] or relevant artefacts from

the project management repository [138]. Thung et al. [243] recommend relevant API methods to assist the implementation of an incoming feature request by analysing request history and textual similarity between API details and the request texts. In short, each of these relevant studies above analyse lexical similarity between a query and the signature or documentation of the API for finding out candidate APIs. Such approaches might not be always effective and might face vocabulary mismatch issues given that choice of query keywords could be highly subjective [83]. On the other hand, we exploit three co-occurrence heuristics that are derived from crowdsourced knowledge, and they are found to be more effective in the selection of candidate API classes. Co-occurrence heuristics overcome the vocabulary mismatch issues [83, 95], and provide a generic, both language and project independent solution. Besides, we exploit the expertise of a large crowd of technical users stored in Stack Overflow for API recommendation which none of the earlier relevant studies did. Zhang et al. [274] determine semantic distance between NL keywords and API classes using a neural network model (CBOW), and suggest relevant API classes for a generic NL query intended for code search. They collect their API classes from the OSS projects whereas ours are collected from Stack Overflow, the largest programming Q & A site on the web. Their work is closely related to ours. We compare with two variants of Thung et al. and Zhang et al., and readers are referred to Sections 7.4.10, 7.4.12 for the detailed comparison. Since Thung et al. outperform Chan et al. as reported [243], we compared with Thung et al. for our validation.

### 7.6.2 API Usage Pattern Recommendation

Thummalapenta and Xie [240] propose *ParseWeb* that takes in a *source object type* and a *destination object type*, and returns a sequence of method invocations that serve as a solution that yields the destination object from the source object. Xie and Pei [264] take a query that describes the method or class of an API, and recommends a frequent sequence of method invocations for the API by analysing hundreds of open source projects. Warr and Robillard [254] recommend a set of API methods that are relevant to a target method by analysing the structural dependencies between the two sets. Each of these techniques is relevant to our work since they recommend API methods. However, they operate on structured queries rather than natural language queries, and thus comparing ours with theirs is not feasible. Of course, we introduced three heuristics and exploited crowd knowledge for API recommendation, which were not considered by any of these existing techniques. This makes our contribution significantly different from all of them.

### 7.6.3 Query Reformulation for Code Search

There have been a number of studies on query reformulation that target either project-specific code search (e.g., concept/feature location [84, 95, 98, 104, 109, 121, 188, 191, 265], bug localization [65, 231]) or general-purpose code search [92, 136, 168]. Gay et al. [84] first propose “relevance feedback” based model for query reformulation in the context of concept location. Once the initial query retrieves search results, a developer is expected to mark them as either relevant or irrelevant. Then their model analyses these marked source

documents, and expands the initial query using *Rocchio expansion* [213]. Although developer feedbacks on document relevance are effective, collecting them is time consuming and sometimes infeasible as well. Therefore, latter studies came up with a less efficient but feasible alternative—*pseudo relevance feedback*—for query reformulation where they consider only Top-K search results (retrieved by the initial query) as the relevant ones. Then they apply term weighting [120, 191, 213], term context analysis [104, 109, 231, 265], query quality analysis [95, 98], and machine learning [98] to reformulate a given query for concept/feature location. Our work falls into the category of general purpose code search. Relevance feedback models were also adopted in this case for query reformulation. Wang et al. [251] incorporate developer feedback in the code search, and improve result ranking. Nie et al. [168] employ Stack Overflow as the provider of relevance feedback on the initial query, and then reformulate it using Rocchio expansion. Although we do not apply relevance feedback for query reformulation, the work of Nie et al. is not only closely related to ours but also relatively more recent. Another closely related recent work by Zhang et al. [274] leverages semantic distance between NL keywords and API classes, and then expands the NL queries using semantically relevant API classes for code search. We thus compare our technique with three techniques above [168, 243, 274], and the detailed comparison can be found in RQ<sub>10</sub>. Li et al. [136] develop a lexical database by using software-specific tags from Stack Overflow questions, and reformulate a given query using synonymy substitution. However, their approach searches for relevant software projects rather than source code segments. Campbell and Treude [58] mine titles from Stack Overflow questions, and suggest automatic expansion to the initial query in the form of auto-completion. However, this approach also relies on textual similarity between initial query and the expanded query, and thus, is subject to the vocabulary mismatch issues. On the contrary, we overcome such issues using three co-occurrence based heuristics. Besides, their approach is constrained by a fixed set of predefined queries from Stack Overflow questions, and thus, might not help much in the formulation of custom queries. RACK does not impose such restrictions on query formulation.

#### 7.6.4 Crowdsourced Knowledge Mining

Existing studies [136, 168, 176, 188, 229, 272] leverage crowd generated knowledge to support several search related activities performed by the developers. Yuan et al. [272] first used programming questions and answers from Stack Overflow to identify semantically similar software specific word pairs. They first construct context of each word by collecting co-occurred words from Stack Overflow questions, answers and tags. Then they determine the semantic similarity between a pair of NL words based on the overlap between their corresponding contexts. Such word pairs might help in addressing the vocabulary mismatch issues with web search. However, they might not help much with code search given that source code and regular texts often hold different semantics for the same word [45, 265]. Wong et al. [259] mine developer’s descriptions of the code snippets from Stack Overflow answers, and suggest them as comments for similar code segments. Rigby and Robillard [211] mine posts from Stack Overflow, and extract salient program elements using regular expressions and machine learning. Along the same line with the earlier studies, we mine Stack Overflow

questions and answers to reformulate a given natural language query for code search. While our work is related to earlier studies [136, 168], it is also significantly different in many ways. First, we suggest relevant API classes for a NL-query by considering keyword-API co-occurrences whereas Nie et al. suggest mostly natural language terms as query expansions by employing pseudo-relevance feedback. Li et al. [136] reformulate queries using crowd wisdom from Stack Overflow for searching open source projects whereas our queries are targeted for more granular software artefacts, e.g., source code snippets. Furthermore, we suggest relevant API classes in contrast with synonymous NL tags by Li et al., which are more appropriate and effective for code search [45]. Another contemporary work [229] uses all program artifacts indiscriminately from Stack Overflow posts for expanding code search queries which could be noisy. On the contrary, we leverage co-occurrences between NL keywords (in the question title) and API classes (in the accepted answer) as a proxy to their relevance, and choose appropriate API classes only for our query reformulation.

Our work in this article also significantly extends our earlier work [201] in various aspects. We improve earlier heuristics by extensively calibrating their weights and thresholds, and introduce a novel heuristic—Keyword Pair API Co-occurrence—that performs better than the earlier ones. We conduct experiments with a relatively larger dataset containing 175 distinct queries, and further evaluate them in terms of relevant code retrieval performance which was missing in the earlier work. We not only compare with several state-of-the-art studies but also demonstrate RACK’s potential for application in the context of traditional web/code search practices. Furthermore, we extend our earlier analysis and answer 11 research questions as opposed to seven questions answered by the earlier work.

## 7.7 Summary

Software developers often search for relevant code examples on the web [55], and reuse them in various software maintenance tasks (e.g., new feature addition). As in the local code searches (e.g., bug localization, concept location), appropriate query construction is also a major challenge in the Internet-scale code search [45]. We propose a novel query reformulation technique—RACK—that accepts a generic query, expands the query with relevant API classes carefully mined from Stack Overflow, and then delivers an improved, reformulated query for Internet-scale code search. Experiments using 175 queries from three tutorial sites show that our reformulated queries (using relevant API classes) can significantly improve upon the given queries in terms of code retrieval performance. Comparison with the state-of-the-art approaches shows that our approach outperforms them in the query reformulation by a significant margin. Furthermore, our technique is generic, project independent, and it exploits invaluable crowd generated knowledge from Stack Overflow for automated query reformulation.

Despite these positive instances above, RACK bears the risk of hurting a given search query with noisy or false-positive API classes. The implicit association between the query keywords and the API classes is an effective proxy to their relevance. However, such a proxy overlooks the underlying semantics of both the keywords and the API classes, which could be a crucial factor. In the next chapter, our sixth study

(NLP2API, Chapter 8) overcomes this issue. NLP2API accepts a programming task description as a query, reformulates the query with relevant API classes that are determined based on the underlying semantics of both the query keywords and the API classes, and then delivers an improved, reformulated query for Internet-scale code search.

## CHAPTER 8

# SEARCH QUERY REFORMULATION FOR INTERNET-SCALE CODE SEARCH USING WORD SEMANTICS

Software maintenance costs a major part of the development time and efforts [88]. Software developers often search for relevant code examples on the web [55], and reuse them in various maintenance tasks (e.g., new feature addition). As in the local code searches (e.g., bug localization, concept location), appropriate query construction is also a major challenge in the Internet-scale code search. Our previous study (RACK, Chapter 7) leverages the implicit association between query keywords and API classes within Stack Overflow Q&A threads as a proxy to their relevance. Although such an association was found reliable as a proxy to relevance (Section 7.4), the underlying semantics of both the keywords and the candidate API classes were overlooked, which could be a crucial factor. In this chapter, we address this issue with another study. Here, we present NLP2API that accepts a programming task description as a query, reformulates the query with relevant API classes by leveraging query-API semantic distance and by mining crowd knowledge from Stack Overflow, and then delivers an improved, reformulated search for Internet-scale code search.

The rest of the chapter is organized as follows—Section 8.1 presents an overview of our study, and Section 8.2 discusses our proposed technique for automatic query reformulation for Internet-scale code search. Section 8.3 discusses our evaluation and validation details, Section 8.4 focuses on the threats to validity, Section 8.5 discusses the related work, and finally Section 8.6 concludes the chapter with future work.

### 8.1 Introduction

Software developers spend about 19% of their development time in searching for relevant code snippets (e.g., API usage examples) on the web [55, 263]. Although open source software repositories (e.g., GitHub, SourceForge) are a great source of such code snippets, retrieving them is a major challenge [44]. Developers often use traditional code search engines (e.g., GitHub native search) to collect code snippets from such repositories using generic natural language queries [45]. Unfortunately, such queries hardly lead to any relevant results (i.e., only 12% [45]) due to vocabulary mismatch issues [83, 142]. Hence, the developers frequently reformulate their queries by removing irrelevant keywords and by adding more appropriate keywords. Studies [45, 120, 219] have shown that 33%–73% of all the queries are incrementally reformulated by the developers. These manual reformulations involve numerous trials and errors, and often cost significant development time

and efforts [120]. One way to help the developers overcome this challenge is to automatically reformulate their generic queries (which are often poorly designed [120, 142]) with meaningful query keywords such as relevant API classes. Our work in the chapter addresses this particular research problem – *query reformulation* targeting *Internet-scale code search*.

Several existing studies offer automatic query reformulation supports for Internet-scale code search using either actual or pseudo relevance feedback on the query [168, 251] and by mining crowd generated knowledge stored in Stack Overflow programming Q & A site [136, 168, 201]. Nie et al. [168] collect pseudo-relevance feedback (PRF) on a given query by employing Stack Overflow as a feedback provider, and then suggest query expansion by analysing the feedback documents, i.e., relevant programming questions and answers. However, they treat the Q & A threads as regular texts, and suggest natural language (i.e., software-specific) terms as query expansion. Existing evidence suggests that queries containing only natural language terms perform poorly in code search [45]. Rahman et al. [201] mine co-occurrences between query keywords (found in the question titles) and API classes (found in the answers) of Stack Overflow, apply two heuristics, and then suggest a set of relevant API classes for a given query. Unfortunately, their heuristics heavily rely on the association between the query keywords and the candidate API classes for relevance estimation, which might always not be sufficient enough. In particular, such heuristics bear the risk of returning the *generic*, *frequent* but *less-relevant* API classes (e.g., `String`, `ArrayList`, `List`) if appropriate filters are not used.

In this chapter, we propose a novel technique—NLP2API—that automatically identifies relevant API classes for a programming task written as a natural language query, and then reformulates the query using these API classes for Internet-scale code search. We first (1) collect candidate API classes for a query from relevant questions and answers of Stack Overflow (i.e., *crowdsourced knowledge*) (Section 8.2.1), and then (2) identify appropriate classes from the candidates using Borda count (Section 8.2.2) and query-API semantic proximity (i.e., *word semantics*) (Section 8.2.3). In particular, we determine semantics of either a keyword or an API class based on their positions within a high dimensional semantic space developed by *fastText* [54] using 1.40 million questions and answers of Stack Overflow. Then we estimate the relevance of the candidate API class to the search query using their semantic proximity measure. Earlier approaches only perform either local context analysis [168, 251] or global context analysis [134, 144, 201]. On the contrary, our technique analyses both local (e.g., PageRank [57]) and global (e.g., semantic proximity) contexts of the query keywords for relevant API class identification and query reformulation. Thus, NLP2API has a higher potential for query reformulation. Besides, opportunistic blending of pseudo-relevance feedback [62, 222], term weighting methods [57, 114], Borda count [275] and *word semantics* [54] also makes our work *novel*.

Table 1 and Fig. 8.1 present a use-case scenario of our technique where a developer is looking for a working code snippet that can convert a colour image to grayscale without losing transparency. First, the developer issues a generic query—“*Convert image to grayscale without losing transparency*”. Then she submits it to *Lucene*, a search engine that is widely used both by contemporary code search solutions such as GitHub native search [10] or ElasticSearch and by academic studies [98, 164, 191]. Unfortunately, the generic natural



```

BufferedImage master = ImageIO.read(new URL(
"http://www.java2s.com/style/download.png"));
BufferedImage gray = new BufferedImage(master.getWidth(),
master.getHeight(), BufferedImage.TYPE_INT_ARGB);

ColorConvertOp op = new ColorConvertOp(
ColorSpace.getInstance(ColorSpace.CS_GRAY), null);
op.filter(master, gray);

ImageIO.write(master, "png", new File("path/to/master"));
ImageIO.write(gray, "png", new File("path/to/gray/image"));

```

**Figure 8.1:** An example code snippet for the programming task– “Convert image to grayscale without losing transparency” – (taken from [9])

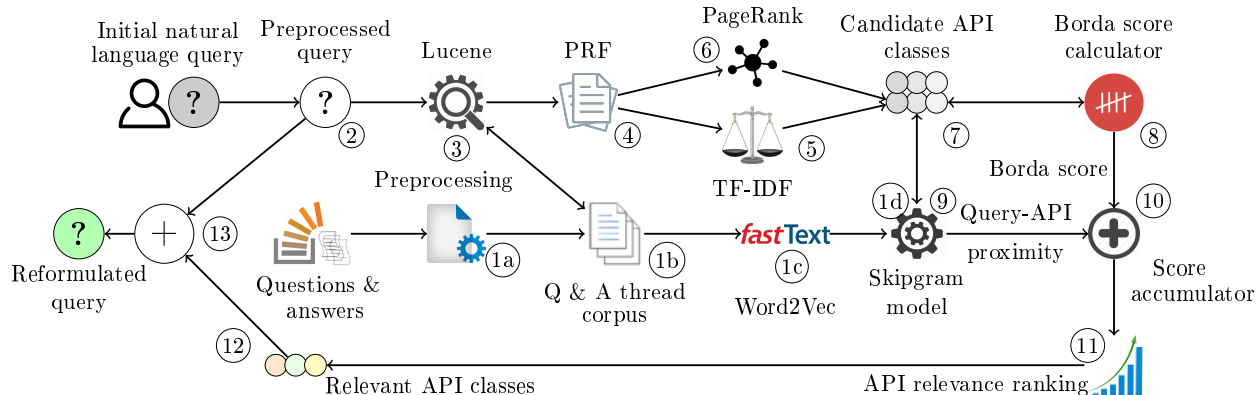
**Table 8.1:** Reformulations of an NL Query for Improved Internet-scale Code Search

Technique	Reformulated Query	QE
Baseline	Convert image to grayscale without losing transparency	115
QECK [168]	{Convert image grayscale losing transparency} + {hsb pixelsByte png iArray img correctly HSB mountainMap enhancedImagePixels file}	11
Google	Convert image to grayscale without losing transparency	02
<b>Proposed</b>	{Convert image grayscale losing transparency} + {BufferedImage Grayscale ImageEdit ColorConvertOp File Transparency ColorSpace BufferedImageOp Graphics ImageEffects}	02

**QE** = Rank of the first correct result returned by the query

language query does not perform well due to vocabulary mismatch between its keywords and the source code, and returns the relevant code snippet (e.g., Fig. 8.1) at the **115<sup>th</sup>** position. On the contrary, (1) our proposed technique *complements* this query with not only *relevant*, but also highly *specific* API classes (e.g., `BufferedImage`, `ColorConvertOp`, `ColorSpace`), and (2) our improved query returns the target code snippet at the *second* position of the ranked list which is a major rank improvement over the baseline. The most recent and closely related technique–QECK [168] returns the same code snippet at the 11<sup>th</sup> position which is not ideal. Google, the most popular web search engine, returns a *similar* code at the second position as well. However, in the case of web search, relevant code snippets are *sporadic* and often buried into a large bulk of unstructured, noisy and redundant natural language texts across multiple web pages which might overwhelm the developer with information overload [151].

Experiments using 310 code search queries randomly collected from four Java tutorial sites–*KodeJava*, *Java2s*, *CodeJava* and *JavaDB*–report that our technique can suggest relevant API classes with 82% Top-10 Accuracy, 48% precision, 58% recall and a reciprocal rank of 0.55 which are 6%, 32%, 48% and 41% higher respectively than those of the state-of-the-art [201]. Comparisons with three state-of-the-art studies and three popular code (or web) search engines – *Google*, *Stack Overflow native search* and *GitHub native search* – reported that our technique (1) can outperform the existing studies [168, 201, 229] in query effectiveness



**Figure 8.2:** Schematic diagram of the proposed query reformulation technique—NLP2API

and (2) can improve upon the precision of these search engines by 17%, 34% and 33% respectively using our reformulated queries. Thus, our work makes the following contributions:

- (a) A novel query reformulation technique—NLP2API—that reformulates generic natural language queries for Internet-scale code search using word semantics and crowd knowledge derived from Stack Overflow.
- (b) Comprehensive evaluation of the proposed technique using 310 queries and validation against the state-of-the-art techniques and widely used web/code search engines.
- (c) A replication package that includes our working prototype and the detailed experimental dataset [18].

## 8.2 NLP2API: Automated Query Reformulation using Word Semantics & Crowd Knowledge for Internet-scale Code Search

Fig. 8.2 shows the schematic diagram of our proposed technique for the reformulation of a generic query targeting Internet-scale code search. Furthermore, Algorithm 9 shows the pseudo-code of our technique. We make use of pseudo-relevance feedback (PRF), crowd generated knowledge stored at Stack Overflow, two term weighting algorithms, and word semantics for our query reformulation as follows:

### 8.2.1 Development of Candidate API Lists

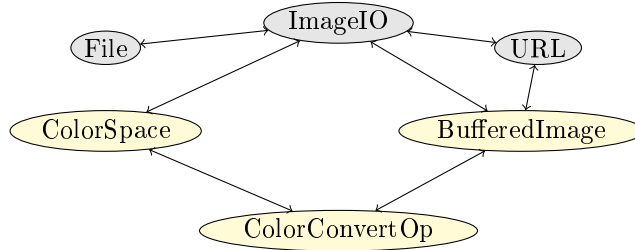
We collect candidate API classes from Stack Overflow Q & A site to reformulate a generic query (i.e., Fig. 8.2, Steps 1a, 1b, 2–7). Stack Overflow is a large body of crowd knowledge with 14 million questions and 22 million answers across multiple programming languages and domains [58]. Hence, it might contain at least a few questions and answers related to any programming task at hand. Earlier studies from the literature [58, 136, 168] also strongly support this conjecture. Given that relevant program elements are a better choice than generic natural language terms for code search [45], we collect API classes as candidates for query reformulation by mining the programming Q & A threads of Stack Overflow.

**Corpus Preparation:** We collect a total of 656,538 Q & A threads related to Java (i.e., using `<java>` tag) from Stack Overflow for corpus preparation (Fig. 8.2, Steps 1a, 1b, Algorithm 9, Line 3). We use the public data dump [35] released on March 2018 for data collection. Since we are mostly interested in the API classes discussed in the Q & A texts, we adopt certain restrictions. First, we make sure that each question or answer contains a bit of code, i.e., the thread is about coding. For this, we check the existence of `<code>` tags in their texts like the earlier studies [76, 80, 177, 198]. Second, to ensure high quality content, we chose only such Q & A threads where the answer was accepted as solution by the person who submitted the question [168, 201]. Once the Q & A threads are collected, we perform standard natural language preprocessing (i.e., removal of stop words, punctuation marks and programming keywords, token splitting) on each thread, and normalize their contents. Given the controversial evidence on the impact of stemming on source code [106], we avoid stemming on these threads given that they contain code segments. Our corpus is then indexed using *Lucene*, a widely used search engine by the literature [98, 164, 191], and later used for collecting feedbacks on a generic natural language query.

**Pseudo-Relevance Feedback (PRF) on the NL Query:** Nie et al. [168] first employ Stack Overflow in collecting pseudo-relevance feedback on a given query. Their idea was to extract software-specific words relevant to a given query, and then to use them for query reformulation. Similarly, we also collect pseudo-relevance feedback on the query using Stack Overflow. We first normalize a natural language query using standard natural language preprocessing (i.e., stopword removal, token splitting), and then use it to retrieve Top-M (e.g.,  $M = 35$ , check  $RQ_1$  for detailed justification) Q & A threads from the above corpus with *Lucene* search engine (i.e., Fig. 8.2, Steps 2–4, Algorithm 9, Lines 4–8). The baseline idea is to extract appropriate API classes from them using appropriate selection methods [139], and then, to use them for query reformulation. We thus extract the program elements (e.g., code segments, API classes) from each of the threads by analysing their HTML contents. We use *Jsoup* [14], a Java library for the HTML scraping. We also develop *two* separate sets of code segments from the questions and answers of the feedback threads. Then we use *two* widely used term-weighting methods –*TF-IDF* and *PageRank*– for collecting candidate API classes from them.

**API Class Weight Estimation with TF-IDF:** Existing studies [84, 98, 168] often apply Rocchio’s method [213] for query reformulation where they use TF-IDF to select appropriate expansion terms. Similarly, we adopt TF-IDF for selecting potential reformulation candidates from the code segments that were collected above. In particular, we extract all API classes from each code segment (i.e., feedback document) with the help of island parsing (i.e., uses regular expressions) [211], and then determine their relative weight (i.e., Fig. 8.2, Step 5, Algorithm 9, Lines 11–12) as follows:

$$TF - IDF(A_i) = (1 + \log(TF_{A_i})) \times \log(1 + \frac{N}{DF_{A_i}}) \quad (8.1)$$



**Figure 8.3:** API co-occurrence graph for code segment in Fig. 8.1

Here  $TF_{A_i}$  refers to total occurrence frequency of an API class  $A_i$  in the collected code segments,  $N$  refers to total Q & A threads in the corpus, and  $DF_{A_i}$  is the number of threads that mentioned API class  $A_i$  in their texts or code segments.

**API Class Weight Estimation with PageRank:** Semantics of a term are often determined by its contexts, i.e., surrounding terms [268, 272]. Hence, inter-dependence of terms is an important factor in the estimation of term weight. However, TF-IDF assumes term independence (i.e., ignores term contexts) in the weight estimation. Hence, it might fail to identify highly important but not so frequent terms from a body of texts [153, 189]. We thus employ another term weighting method that considers dependencies among terms in the weight estimation. In particular, we apply PageRank algorithm [57, 153] to the relevance feedback documents, i.e., relevant code segments, and identify the important API classes as follows:

*Construction of API Co-occurrence Graph:* Since PageRank algorithm operates on a graph-based structure, we transform pseudo-relevance feedback documents into a graph of API classes (i.e., Fig. 8.2, Step 6, Algorithm 9, Line 13). In particular, we extract all API classes from each code segment using island parsing [211], and then develop an ordered list by preserving their initialization order in the code. For example, the code snippet in Fig. 8.1 is converted into a list of six API classes. Co-occurrences of items in a certain context has long been considered as an indication of relatedness among the items [153, 272]. We thus capture the immediate co-occurrences of API classes in the above list, consider such co-occurrences as connecting edges, and then develop an API co-occurrence graph (e.g., Fig. 8.3). We repeat the same step for each of the code segments, and update the connectivities in the graph. We develop one graph for the code segments from questions and another graph for the code segments from answers which were returned as a part of the pseudo-relevance feedback.

*API Class Rank Calculation:* PageRank has been widely used for web link analysis [57] and term weighting in Information Retrieval domain [153]. It applies the underlying mechanism of recommendation or voting for determining importance of an item (e.g., web page, term) [191]. That is, PageRank considers a node as important only if it is recommended (i.e., connected to) by other important nodes in the graph. The same idea has been widely used for separating legitimate pages from spam pages [169]. Similarly, in our problem context, if an API class co-occurs with other important API classes across multiple code segments that are relevant to a programming task, then this API class is also considered to be important for the task. We apply PageRank algorithm on each of the two graphs (i.e., Fig. 8.2, Step 6, Algorithm 9, Line 14),

and determine the importance  $ACR(v_i)$  (i.e., API Class Rank) of each node  $v_i$  by recursively applying the following equation:

$$ACR(v_i) = (1 - \phi) + \phi \sum_{j \in In(v_i)} \frac{ACR(v_j)}{|Out(v_j)|} \quad (0 \leq \phi \leq 1) \quad (8.2)$$

Here,  $In(v_i)$  refers to nodes providing inbound links (i.e., votes) to node  $v_i$  whereas  $Out(v_i)$  refers to nodes that  $v_i$  is connected to through outbound links, and  $\phi$  is the damping factor. In the context of world wide web, Brin and Page [57] considered  $\phi$  as the probability of visiting a web page and  $1 - \phi$  as the probability of jumping off the page by a random surfer. We use a value  $\phi = 0.85$  for our work like the previous studies [57, 153, 189]. We initialize each node with a score of 0.25, and run an *iterative version* of PageRank on the graph. The algorithm pulls out weights from the surrounding nodes recursively, and updates the weight of a target node. This recursive process continues until the scores of the nodes converge below a certain threshold (e.g., 0.0001 [153]) or total iteration count reaches the maximum (e.g., 100 [153]). Once the computation is over, each node (i.e., API class) is left with a score which is considered as a numerical proxy to its relative importance among all nodes.

**Selection of Candidate API Classes:** Once two weights –TF-IDF and PageRank– of each of the potential candidates are calculated, we rank the candidates according to their weights. Then we select Top-N (e.g.,  $N = 16$ , check RQ<sub>1</sub> for justification) API classes from each of the four lists (i.e., two lists for each term weight, Fig. 8.2, Step 7, Algorithm 9, Lines 9–16). In Stack Overflow Q & A site, a question often describes a programming problem (or a task) whereas the answer offers a solution. Thus, API classes involved with the problem and API classes forming the solution should be treated differently for identifying the *relevant* and *specific* API classes for the task. We leverage this inherent differences of context and semantics between questions and answers, and treat their code segments separately unlike the earlier study of Nie et al. [168] that overlooks such differences.

## 8.2.2 Borda Score Calculation

Borda count is a widely used election method where the voters sort their political candidates on a scale of preference [3, 275]. In the context of Software Engineering, Holmes and Murphy [107] first apply Borda count to recommend relevant code examples for the code under development in the IDE. They apply this method to six ranked list of code examples collected using six structural heuristics, and then suggest the most frequent examples across these lists as the most relevant ones. Similarly, we apply this method to our four candidate API lists (i.e., Fig. 8.2, Step 8, Algorithm 9, Lines 22–23) where each of the API classes are ranked based on their importance estimates (e.g., TF-IDF, API Class Rank). We calculate Borda score  $S_B$  for each of the API classes ( $\forall A_i \in A$ ) from the these ranked candidate lists– $WRC = \{WC_Q, WC_A, RC_Q, RC_A\}$ –as follows:

---

**Algorithm 9** Automated Query Reformulation using Relevant API Classes

---

```
1: procedure NLP2API( $Q$ ) ▷  $Q$ : natural language query
2:    $R \leftarrow \{\}$  ▷  $R$ : Relevant API classes
3:    $C \leftarrow \text{developQ\&ACorpus}(SODump)$  ▷  $C$ : SO corpus
4:    $Q_{pp} \leftarrow \text{preprocess}(Q)$ 
5:   ▷ collecting pseudo-relevance feedback
6:    $PRF \leftarrow \text{getPRF}(Q_{pp}, C)$ 
7:    $PRF_Q \leftarrow \text{getQuestionCodeSegments}(PRF)$ 
8:    $PRF_A \leftarrow \text{getAnswerCodeSegments}(PRF)$ 
9:   ▷ collecting candidate API list
10:  for PRF  $prf \in \{PRF_Q, PRF_A\}$  do
11:     $TW \leftarrow \text{calculateTFIDF}(prf, C)$ 
12:     $WC[prf] \leftarrow \text{getTopKWeightedClasses}(TW)$ 
13:     $G \leftarrow \text{developAPICo-occurrenceGraph}(prf)$ 
14:     $ACR \leftarrow \text{calculateAPIClassRank}(G)$ 
15:     $RC[prf] \leftarrow \text{getTopKRankedClasses}(ACR)$ 
16:  end for
17:  ▷ training the fastText model
18:   $M_{ft} \leftarrow \text{getFastTextModel}(\text{preprocess}(SODump))$ 
19:  ▷ API relevance estimation
20:   $A \leftarrow \text{getAllCandidateAPIClasses}(RC \cup WC)$ 
21:  for CandidateAPIClass  $A_i \in A$  do
22:    ▷ calculate Borda score
23:     $S_B[A_i] \leftarrow \text{getBordaScore}(A_i, RC, WC)$ 
24:    ▷ semantic relevance between API class and query
25:     $S_P[A_i] \leftarrow \text{getQuery-APIProximity}(A_i, Q_{pp}, M_{ft})$ 
26:     $R[A_i].score \leftarrow S_B[A_i] + S_P[A_i]$ 
27:  end for
28:  ▷ ranking of the API classes
29:   $rankedClasses \leftarrow \text{sortByFinalScore}(R)$ 
30:  ▷ reformulation of the initial query
31:  return  $Q_{pp} + rankedClasses$ 
32: end procedure
```

---

$$S_B(A_i \in A) = \sum_{RL_j \in WRC} 1 - \frac{\text{rank}(A_i, RL_j)}{|RL_j|} \quad (8.3)$$

Here,  $A$  refers to the set of all API classes extracted from the ranked candidate lists  $WRC$ ,  $|RL_j|$  denotes each list size, and  $\text{rank}(A_i, RL_j)$  returns the rank of class  $A_i$  in the ranked list. Thus, an API class that occurs at the top positions in multiple candidate lists is likely to be more important for a target programming task than the ones that either occurs at the lower positions or does not occur in multiple lists.

### 8.2.3 Query-API Semantic Proximity Analysis

Pseudo-relevance feedback, PageRank (Section 8.2.1) and Borda count (Section 8.2.2) analyse local contexts of the query keywords within a set of tentatively relevant documents (i.e., Q & A threads) and then extract candidate API classes for query reformulation. Although local context analysis is useful, existing studies report that such analysis alone might cause topic drift from the original query [62, 136]. We thus further analyse global contexts of the query keywords, and determine the semantic proximity between the given natural language query and the candidate API classes as follows:

**Word2Vec Model Development:** Mikolov et al. [156] and colleagues propose a neural network based tool—*word2vec*—for learning word embeddings from an ultra-large body of texts where they employ continuous bag of words (CBOW) and skip-gram models. While other studies attempt to define context of a word using co-occurrence frequencies or TF-IDF [150, 201, 272], they offer a probabilistic representation of the context. In particular, they learn *word embeddings* (Section 2.7) for each of the words from the corpus, and map each word to a point in the semantic space so that semantically similar words appear in the close proximity. We leverage this notion of *semantic proximity*, and determine the relevance of a candidate API class to the given query. It should be noted that such proximity measure could be an effective tool to overcome the *vocabulary mismatch issues* [83]. We thus develop a *word2vec* model where 1.3 million programming questions and answers (i.e., 656,538 Q & A pairs, collected in Section 8.2.1) are employed as the corpus. We normalize each question and answer using standard natural language preprocessing, and learn the word embeddings (Fig. 8.2, Step 1b, 1c, 1d, Algorithm 9, Lines 17–18) using skip-gram model. For our learning, we use *fastText* [54], an improved version of *word2vec* that incorporates sub-word information into the model. We performed the learning offline and it took about one hour. It should be noted that our model is learned using default parameters (e.g., output vector size = 100, context window size = 5, minimum word occurrences = 5) provided by the tool.

**Semantic Relevance Estimation:** While a given query contains multiple keywords, a candidate API class might not be semantically close to all of them. We thus capture the maximum proximity estimate between an API class and any of the query keywords as the relevance estimate of the class. In particular, we collect word embeddings (i.e., a vector of 100 real valued estimates of the contexts) of each candidate API

class  $A_i \in A$  and each keyword  $q \in Q$ , and determine their semantic proximity  $S_P$  using *cosine similarity* (i.e., Fig. 8.2, Step 9, Algorithm 9, Lines 24–25) as follows:

$$S_P(A_i \in A) = \{f(A_i, q) \mid f(A_i, q) > f(A_i, q_0) \forall q_0 \in Q\} \quad (8.4)$$

$$f(A_i, q) = \text{cosine}(\text{fastText}(A_i), \text{fastText}(q)) \quad (8.5)$$

Here  $\text{fastText}(\cdot)$  returns the learned word embeddings of either a query keyword or an API class, and  $f(A_i, q)$  returns the *cosine similarity* between their word embeddings. We use `print-word-vectors` option of *fastText*, and collect the word embeddings from our learned model on Stack Overflow.

### 8.2.4 API Class Relevance Ranking & Query Reformulation

Once Borda score  $S_B$  and semantic proximity score  $S_P$  are calculated, we normalize both scores between 0 and 1, and then sum them up using a *linear combination* (i.e., Line 26, Algorithm 9) for each of the candidate API classes. While fine tuned relative weight estimation for these two scores could have been a better approach, we keep that as a part of future work. Besides, equal weights also reported pretty good results (e.g., 82% Top-10 accuracy) according to our investigation. The API classes are then ranked according to their final scores, and Top-K (e.g.,  $K = 10$ ) classes are suggested as the relevant classes for the programming task stated as a generic query (i.e., Fig. 8.2, Steps 10–12, Algorithm 9, Lines 19–29). These API classes are then appended to the given query as reformulations [98] (i.e., Fig. 8.2, Steps 13, Algorithm 9, Lines 30–31). Table 8.1 shows our reformulated query for the showcase natural language query using the relevant API classes suggested by NLP2API.

## 8.3 Experiment

We conduct experiments with 310 code search queries randomly collected from four popular programming tutorial sites, and evaluate our query reformulation technique. We choose five appropriate performance metrics from the literature, and evaluate two aspects of our provided supports-(1) relevant API class suggestion and (2) query reformulation. Our technique is also validated against three state-of-the-art techniques [168, 201, 229] and three popular code/web search engines including Google. We thus answer five research question using our experiments as follows:

- **RQ<sub>1</sub>**: How does NLP2API perform in recommending relevant API classes for a given query? How do different parameters and thresholds influence the performance?
- **RQ<sub>2</sub>**: Can NLP2API outperform the state-of-the-art technique on relevant API class suggestion for a query?
- **RQ<sub>3</sub>**: Can the reformulated queries of NLP2API outperform the baseline natural language queries?



- **RQ<sub>4</sub>**: Can NLP2API outperform the state-of-the-art technique on query reformulation that uses crowd-sourced knowledge from Stack Overflow?
- **RQ<sub>5</sub>**: Can our approach, NLP2API, significantly improve the results provided by state-of-the-art code or web search engines?

### 8.3.1 Experimental Dataset

**Dataset Collection:** We collect 310 code search queries from four popular programming tutorial sites—KodeJava [15], Java2s [11], CodeJava [7] and JavaDB [13]—for our experiments. While 150 of these queries were taken from a publicly available dataset [201], we attempted to extend the dataset by adding 200 more queries. However, after removing the duplicates and near duplicates, we ended up with 160 queries. Thus, our dataset contains a total of 310 (i.e., 150 old + 160 new) search queries. Each of these sites above discusses hundreds of programming tasks as Q & A threads where each thread generally contains (1) a question title, (2) a solution (i.e., code), and (3) a prose explaining the code succinctly. The question title (e.g., “*How do I decompress a GZip file in Java?*” [20]) generally comprises of a few important keywords and often *resembles* a real life search query. We thus use these titles from tutorial sites as code search queries in our experiments, as were also used by the earlier studies [63, 201].

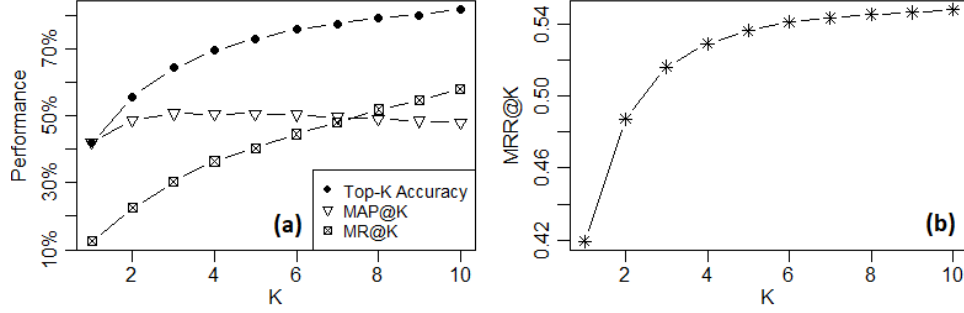
**Ground Truth Preparation:** The prose that explains code in the tutorial sites above often includes one or more API classes from the code (e.g., `GZipInputStream`, `FileOutputStream`). Since these API classes are chosen to explain the code that implements a programming task, they are generally relevant and specific to the task. We thus consider these *relevant* and *specific* API classes as the *ground truth* for the corresponding question title (i.e., our search query) [201]. We develop a *ground truth API set* to evaluate the performance of our technique in the API class suggestion. We also collect the code segments from each of the 310 Q & A threads from the tutorial sites above as the *ground truth code segments*, and use them to evaluate the query reformulation performance (i.e., in terms of code retrieval) of our technique. Given that these API classes and code segments are publicly available online and were consulted by thousands of technical users over the years, subjectivity associated with their relevance to the corresponding tasks (i.e., our selected queries) is minimized [63]. Our dataset preparation step took  $\approx$  **25** man hours.

**Replication Package:** Our dataset, working prototype and other materials are *accepted* for publication [195]. They are publicly available [18] for replication and third party reuse.

### 8.3.2 Performance Metrics

We choose five performance metrics that were widely adopted by relevant literature [63, 152, 168, 191, 201, 243], for the evaluation and validation of our technique as follows:

**Top-K Accuracy / Hit@K:** It is the percentage of search queries for each of which at least one item (e.g., API class) from the *ground truth* is returned within the Top-K results [239, 243, 250].



**Figure 8.4:** Performance of NLP2API in API class suggestion for various Top-K results

**Table 8.2:** Performance of NLP2API in Relevant API Suggestion

Performance Metric	Top-1	Top-3	Top-5	Top-10
Top-K Accuracy	41.94%	64.19%	<b>72.90%</b>	<b>81.61%</b>
Mean Reciprocal Rank@K	0.42	0.52	<b>0.54</b>	<b>0.55</b>
Mean Average Precision@K	41.94%	<b>50.62%</b>	<b>50.56%</b>	47.85%
Mean Recall@K	12.53%	30.17%	<b>40.28%</b>	<b>57.87%</b>

**Top-K** = Performance measures for Top-K suggestions

**Mean Reciprocal Rank@K (MRR@K):** Reciprocal Rank@K is defined as the multiplicative inverse of the rank of first relevant item (e.g., API class from ground truth) in the Top-K results returned by a technique [220, 276]. Mean Reciprocal Rank@K (MRR@K) averages such measures for all queries.

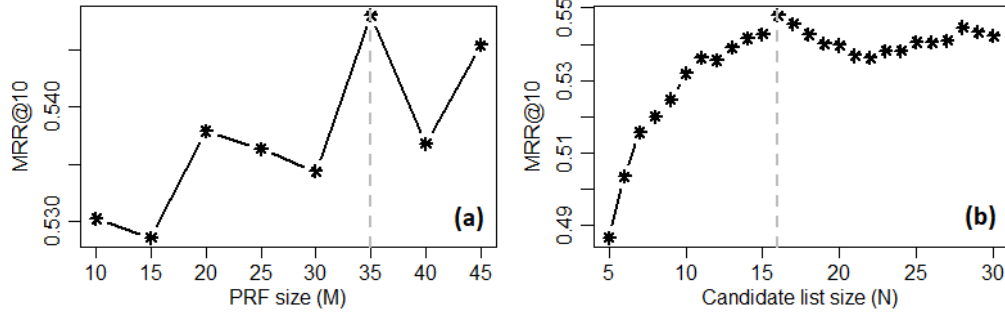
**Mean Average Precision@K (MAP@K):** Precision@K is the precision calculated at the occurrence of  $K^{th}$  item in the ranked list. Average Precision@K (AP@K) averages the precision@K for all relevant items (e.g., API class from ground truth) within the Top-K results for a search query [220, 276]. Mean Average Precision@K is the mean of Average Precision@K for all queries from the dataset.

**Mean Recall@K (MR@K):** Recall@K is defined as the percentage of ground truth items (e.g., API classes) that are correctly recommended for a query in the Top-K results by a technique [63, 249]. Mean Recall@K (MR@K) averages such measures for all queries from the dataset.

**Query Effectiveness (QE):** It is defined as the rank of first correct item (i.e., ground truth code segment) in the result list returned by a query. The measure is an approximation of the developer’s effort in locating the first code segment relevant to a given query. Thus, the lower the effectiveness measure is, the more effective the query is [98, 164, 191]. We use this measure to evaluate the improvement of a query through reformulations offered by a technique.

### 8.3.3 Evaluation of NLP2API: Relevant API Class Suggestion

We first evaluate the performance of our technique in the relevant API class suggestion for a generic code search query. We make use of 310 code search queries (Section 8.3.1) and four performance metrics (Section 8.3.2) for this experiment. We collect Top-K (e.g.,  $K=10$ ) API classes suggested for each query, compare



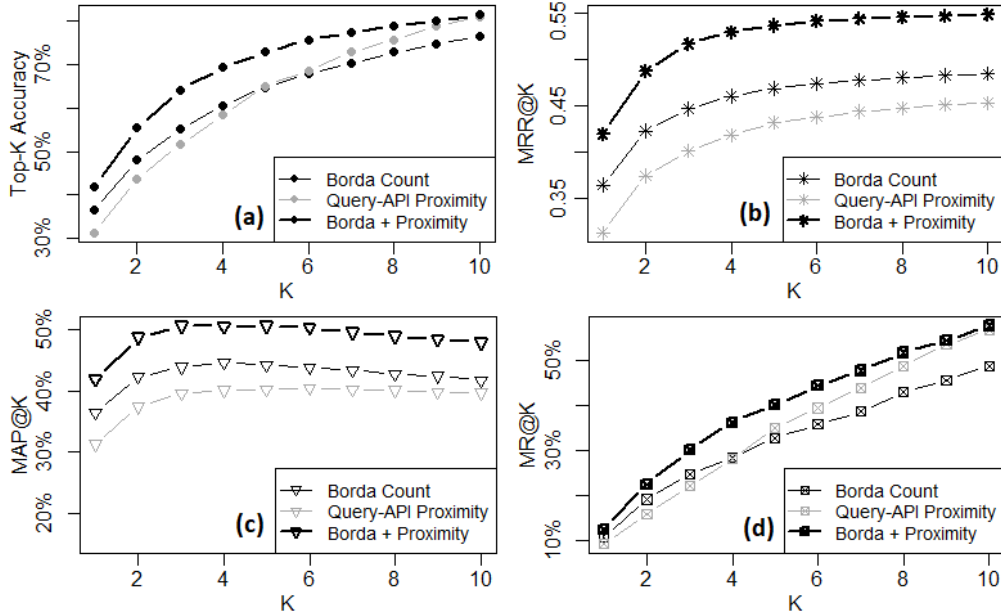
**Figure 8.5:** Impact of (a) PRF size ( $M$ ), and (b) Candidate API list size ( $N$ ) on relevant API class suggestion from Stack Overflow

them with the *ground truth API classes*, and then determine our API suggestion performance. In this section, we also answer RQ<sub>1</sub> and RQ<sub>2</sub> as follows:

**Answering RQ<sub>1</sub>—Relevant API Class Suggestion:** From Table 8.2, we see that our technique returns relevant API classes for 73% of the queries with 51% mean average precision and 40% recall when only Top-5 results are considered. That is, half of the suggested classes come from the *ground truth*, and our approach succeeds for seven out of 10 queries. More importantly, it achieves a mean reciprocal rank of 0.54. That means, on average, the first relevant API class can be found at the second position of the result list. Such classes can also be found at the first position for 42% of the queries. All these statistics are highly promising according to relevant literature [201, 243]. Fig. 8.4 further demonstrates our performance measures for Top-1 to Top-10 results. We see that accuracy, recall and reciprocal rank measures increase monotonically which are expected. Interestingly, the precision measure shows an almost steady behaviour. That means, as more results were collected, our technique was able to *filter out* the *false positives* which demonstrates its high potential for API suggestion.

**Impact of Pseudo-Relevance Feedback Size ( $M$ ) and Candidate API List Size ( $N$ ):** We investigate how different sizes of pseudo-relevance feedback (i.e., number of Q & A threads retrieved from Stack Overflow by the given query) and candidate API list (i.e., detailed in Section 8.2.1) affect the performance of our technique. We conduct experiments using 10–45 feedback Q & A threads and 5–30 candidate API classes. We found that these parameters improved accuracy and recall measures monotonically (i.e., as expected) but affected precision measures in an irregular fashion (i.e., not monotonic). However, we found an interesting pattern with mean reciprocal rank. From Fig. 8.5, we see that mean reciprocal rank@10 of our technique reaches the maximum when (a) pseudo-relevance feedback size,  $M$  is 35 and (b) candidate API list size,  $N$  is 16. We thus adopt these thresholds, i.e.,  $M = 35$  and  $N = 16$ , in our technique for the experiments.

**Borda Count vs. Query-API Class Proximity as API Relevance Estimate:** Once candidate API classes are selected (Section 8.2.1), we employ two proxies (Sections 8.2.2, 8.2.3) for estimating the relevance of an API class to the NL query. We compare the appropriateness of these proxies— *Borda Count* and *Query-API Proximity*— in capturing the API class relevance, and report our findings in Fig. 8.6. We see that Borda Count is more effective than Query-API Proximity in capturing the relevance of an API



**Figure 8.6:** Comparison between Borda count and Query-API proximity in estimating API relevance using (a) accuracy, (b) reciprocal rank, (c) precision, and (d) recall

class to a given query. However, the proximity demonstrates its potential especially with accuracy and recall measures. More interestingly, combination of these two proxies ensures the best performance of our technique in all four metrics. Non-parametric statistical tests also report that performances with Borda+Proximity are significantly higher than those with either Borda Count (i.e., all  $p$ -values  $< 0.05$ ,  $0.34 \leq \Delta \leq 0.82$  (*large*)) or Query-API Semantic Proximity (i.e., all  $p$ -values  $< 0.05$ ,  $0.20 \leq \Delta \leq 0.90$  (*large*)).

We also investigate the parameters of *fastText* [54] that were used to determine the query-API proximity. Although we experimented using various custom parameters, we did not see any significant performance gain over the default parameters. Besides, increased thresholds (e.g., context window size, output vector size) could be computationally costly. We thus adopt the default settings of *fastText* in this work.

**Summary of RQ<sub>1</sub>:** Our technique provides the first relevant API class at the *second* position,  $\approx 50\%$  of our suggested classes are true positive, and the technique succeeds *eight* out of 10 times (i.e., **82%** Top-10 accuracy). Besides, our adopted parameters and thresholds (e.g.,  $M$ ,  $N$ ) are justified.

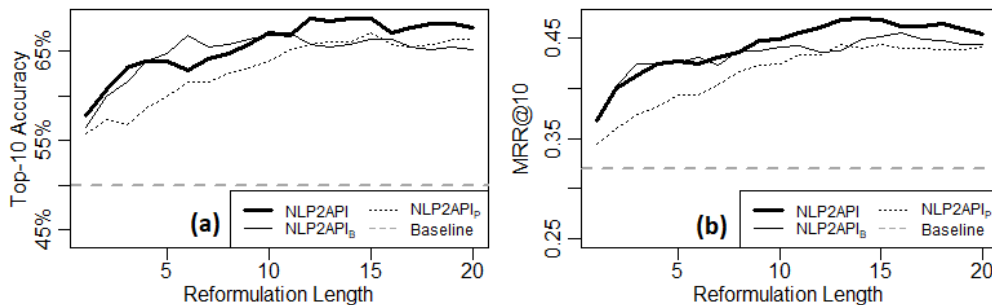
**Answering RQ<sub>2</sub>– Comparison with Existing Studies on Relevant API Class Suggestion:** We compare our technique with the state-of-the-art approach – RACK [201] – on API class suggestion for a natural language query. Rahman et al. [201] employ two heuristics– Keyword-API Co-occurrence (KAC) and Keyword-Keyword Coherence (KKC)–for suggesting relevant API classes from Q & A threads of Stack Overflow for a given query. Their approach outperformed earlier approaches [63, 243] which made it the state-of-the-art in relevant API class suggestion. We collect the authors’ implementation of RACK from corresponding web portal, ran the tool as is on our dataset, and then extract the evaluation results.

From Table 8.3, we see that our technique–NLP2API– outperforms RACK especially in precision, recall and reciprocal rank. It should be noted that our reported performance measures for RACK are pretty close

**Table 8.3:** Comparison with the State-of-the-art in API Class Suggestion

Technique	Metric	Top-1	Top-3	Top-5	Top-10
RACK [201]	Top-K Accuracy	20.97%	52.90%	64.19%	77.10%
	MRR@K	0.21	0.35	0.37	0.39
	MAP@K	20.97%	34.76%	36.76%	36.38%
	MR@K	6.25%	20.81%	28.06%	39.22%
NLP2API (Proposed)	Top-K Accuracy	41.94%	64.19%	<b>72.90%</b>	<b>81.61%</b>
	MRR@K	0.42	0.52	<b>0.54</b>	<b>0.55</b>
	MAP@K	41.94%	<b>50.62%</b>	<b>50.56%</b>	47.85%
	MR@K	12.53%	30.17%	<b>40.28%</b>	<b>57.87%</b>

**Top-K** = Performance measures for Top-K suggestions

**Figure 8.7:** Reformulated vs. baseline query using (a) Top-10 accuracy and (b) MRR@10

to the authors’ reported measures [201], which indicates a *fair comparison*. We see that RACK recommends API classes correctly for 64% of the queries with 37% precision, 28% recall and a reciprocal rank of 0.37 when Top-5 results are considered. On the contrary, our technique recommends correctly for 73% of the queries with 51% precision, 40% recall and a promising reciprocal rank of 0.54 in the same context. These are 14%, 38%, 44% and 46% improvement respectively over the state-of-the-art performance measures. Statistical tests for various Top-K results (i.e.,  $1 \leq K \leq 10$ ) also reported significance (i.e., all  $p\text{-values} \leq 0.05$ ) of our technique over the state-of-the-art with large effect sizes (i.e.,  $0.39 \leq \Delta \leq 0.90$ ).

**Summary of RQ<sub>3</sub>:** Our technique outperforms the state-of-the-art approach on relevant API class suggestion, and it suggests relevant API classes with **38%** higher precision and **46%** higher reciprocal rank than those of the state-of-the-art.

### 8.3.4 Evaluation of NLP2API: Query Reformulation

Although our approach outperforms the state-of-the-art on relevant API class suggestion, we further apply the suggested API classes to query reformulations. Then we demonstrate the potential of our reformulated queries for improving the code snippet search. In this section, we also answer RQ<sub>3</sub>, RQ<sub>4</sub> and RQ<sub>5</sub> using our experiments as follows:

**Table 8.4:** Impact of Reformulations on Generic NL Queries

Reformulation	RL	Improved/MRD	Worsened/MRD	Preserved
NLP2API <sub>B</sub>	05	43.23%/-245	31.29%/+54	25.48%
	10	<b>48.07%</b> /-223	26.13%/+65	25.81%
NLP2API <sub>P</sub>	10	40.97%/-148	30.97%/+44	28.06%
NLP2API	05	40.00%/-159	27.74%/+54	32.26%
	10	<b>48.07%</b> /-209	25.16%/+45	<b>26.77%</b>
	15	<b>49.03%</b> /-217	<b>22.26%</b> /+46	<b>28.71%</b>

MRD = Mean Rank Difference between reformulated and given queries

### Answering RQ<sub>3</sub>—Improvement of Natural Language Queries with the Suggested API Classes:

We reformulate each of the generic natural language queries for code search using the API classes suggested by our technique. Then we investigate the performance of these reformulated queries using code search. We prepare a code corpus of 4,170 code segments where 310 segments are *ground truth* code segments (Section 8.3.1) and 3,860 code segments were taken from a publicly available and curated dataset [184] based on hundreds of GitHub projects. We normalize these segments using standard natural language preprocessing (i.e., stop and keyword removal, token splitting), and index them with *Lucene*. We then perform code search on this corpus, and contrast between generic natural language queries and our reformulated queries in terms of their Effectiveness and code retrieval performances.

From Table 8.4, we see that our reformulations improve or preserve 75% (i.e., 48% improvement and 27% preserving) of the given queries. The improvement ratio reaches the maximum of 49% with a reformulation length of 20. According to relevant literature [98, 164, 191], such statistics are promising. Fig. 8.7 further demonstrates the impact of our reformulations on the baseline generic queries. We see that the baseline natural language queries retrieve ground truth code segments with 50% Top-10 accuracy (dashed line, Fig. 8.7-(a)) and 0.32 mean reciprocal rank (dashed line, Fig. 8.7-(b)). On the contrary, our reformulated queries achieve a maximum of 69% Top-10 accuracy with a reciprocal rank of 0.47 which are 37% and 47% higher respectively than the baseline. Quantile analysis in Table 8.5 also shows that our provided result ranks are more promising than those of the baseline queries.

**Summary of RQ<sub>3</sub>:** Reformulations offered by our technique improve **49%** of the generic natural language queries, and the reformulated queries achieve **37%** higher accuracy and **47%** higher reciprocal rank than those of the generic NL queries.

### Answering RQ<sub>4</sub>—Comparison with Existing Query Reformulation Techniques:

Nie et al. [168] collect pseudo-relevance feedbacks from Stack Overflow on a given query and then apply Rocchio’s method to expand the query. Their approach, QECK, outperformed earlier studies [144, 152] on query reformulation targeting code search which made it the state-of-the-art. Another contemporary work, CoCaBu [229] applies Vector Space Model (VSM) in identifying appropriate program elements from Stack Overflow posts. To the best of our knowledge, these are the most recent and most closely related works to ours. Due to the

**Table 8.5:** Comparison of Query Effectiveness with Existing Query Reformulation Techniques

Technique	#QC	Improvement						Worsening						Preserving		
		#Improved	Mean	Q1	Q2	Q3	Min.	Max.	#Worsened	Mean	Q1	Q2	Q3	Min.	Max.	#Preserved
QECK [168]	310	72 (23.23%)	139	02	11	74	01	1,861	177 (57.10%)	131	11	35	163	02	1,259	61 (19.68%)
<b>RACK</b> [201]	310	105 (33.87%)	75	02	08	60	01	971	147 (47.42%)	136	07	31	156	02	1,277	58 (18.71%)
<b>CoCaBu</b> [229]	310	<b>113 (36.45%)</b>	191	02	14	103	01	2,607	<b>131 (42.26%)</b>	102	06	24	91	02	1,567	66 (21.29%)
Baseline	310	-	-	07	25	145	02	1,460	-	-	01	03	15	01	582	-
<b>NLP2API</b>	310	<b>149 (48.07%)</b>	<b>170</b>	<b>02</b>	<b>12</b>	<b>74</b>	<b>01</b>	2,816	<b>78 (25.16%)</b>	<b>75</b>	<b>03</b>	<b>13</b>	<b>59</b>	<b>02</b>	<b>826</b>	<b>83 (26.77%)</b>
<b>NLP2API<sub>max</sub></b>	310	<b>152 (49.03%)</b>	<b>172</b>	<b>02</b>	<b>10</b>	<b>61</b>	<b>01</b>	2,926	<b>69 (22.26%)</b>	<b>73</b>	<b>03</b>	<b>11</b>	<b>70</b>	<b>02</b>	<b>786</b>	<b>89 (28.71%)</b>

**Mean** = Mean rank of first correct results returned by the queries,  $Q_i = i^{th}$  quartile of all ranks considered

unavailability of authors’ prototype, we re-implement them ourselves using their best performing parameters (e.g., PRF size = 5–10, reformulation length = 10), and then compare them with ours. We also compare with RACK [201] in the context of query reformulation due to its highly related nature.

Table 8.5 shows a quantile analysis of the result ranks provided by the existing techniques. If results are returned closer to the top of the list by a reformulated query than its baseline counterpart, we call it *query improved* and vice versa as *query worsened*. We see that CoCaBu and RACK perform relatively higher than QECK. CoCaBu improves 36% and worsens 42% of the 310 baseline queries. On the contrary, our technique improves 48% and worsens 25% of the given queries which are 32% higher and 40% lower respectively than those of CoCaBu. Furthermore, according to the quantile analyses, the extents of our rank improvement over the baseline are comparatively higher than the extents of rank worsening which indicates a net benefit of the reformulation operations.

**Summary of RQ<sub>4</sub>:** Our technique outperforms the state-of-the-art approaches on query reformulation, and it improves **32%** more and worsens **40%** less queries than those of the state-of-the-art.

**Answering RQ<sub>5</sub>–Comparison with Existing Code/Web Search Engines:** Although our approach outperforms the state-of-the-art studies [168, 201] on relevant API suggestion and query reformulation, we further compare with two popular web search engines – *Google*, *Stack Overflow native search* – and one popular code search engine – *GitHub code search*. Given the enormous and *dynamic index database* and *restrictions* on the *query length* or *type*, a full scale or direct comparison with these search engines is neither feasible nor fair. We thus investigate whether results returned by these contemporary search engines for generic queries could be significantly improved or not with the help of our reformulated queries.

**Collection of Search Results and Establishment of Ground Truth:** We first collect Top-30 results returned by each search engine for each of the 310 queries. For result collection, we make use of *Google’s custom search API* [8] and the native API endpoints provided by Stack Overflow and GitHub. Since our goal is to find relevant code snippets, we adopt a pragmatic approach in the establishment of ground truth for this experiment. In particular, we analyse those 30 results semi-automatically, look for *ground truth code segments* (i.e., collected in Section 8.3.1) in their contents, and then select Top-10 results as *ground truth search results* that contain either the ground truth code or highly similar code. It should be noted that ground truth code segments and our suggested API classes are taken from two different sources.

**Comparison between Initial Search Results and Re-ranked Results with Reformulated Queries:** While the search engines return results mostly for the natural language queries, we further re-rank the results with our reformulated queries (i.e., generic search keywords + relevant API classes) using lexical similarity analysis (e.g., cosine similarity [184]). We then evaluate Top-10 results both by each search engine and by our re-ranking approach against the *ground truth search results*, and demonstrate the potential of our reformulations.

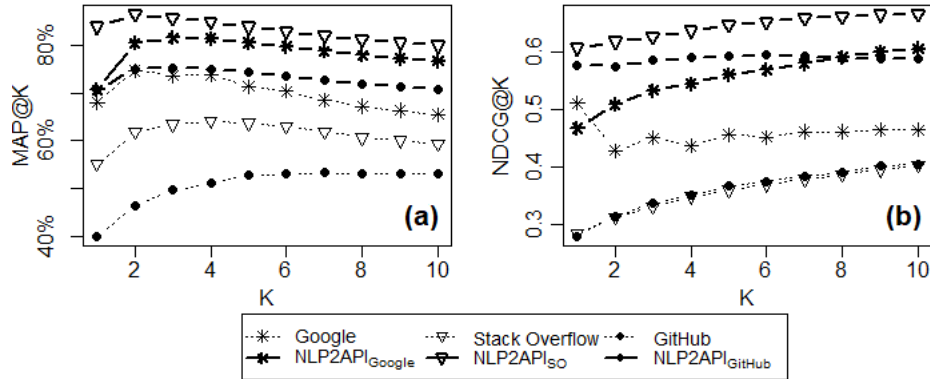
From Table 8.6, we see that the re-ranking approach that leverages our reformulated queries improves the initial search results returned by each of the engines. In particular, the performances are improved in terms



**Table 8.6:** Comparison with Popular Web/Code Search Engines

Technique	Hit@10	MAP@10	MRR@10	NDCG@10
Google	100.00%	65.50%	0.80	0.47
<b>NLP2API<sub>Google</sub></b>	100.00%	<b>76.73%</b>	<b>0.83</b>	<b>0.61</b>
Stack Overflow	90.65%	59.46%	0.67	0.40
<b>NLP2API<sub>SO</sub></b>	91.29%	<b>79.95%</b>	<b>0.87</b>	<b>0.67</b>
GitHub	88.06%	53.06%	0.55	0.41
<b>NLP2API<sub>GitHub</sub></b>	89.03%	<b>70.69%</b>	<b>0.78</b>	<b>0.59</b>

NDCG=Normalized Discounted Cumulative Gain [253]

**Figure 8.8:** Comparison between popular web/code search engines and NLP2API in relevant code segment retrieval using (a) MAP@K and (b) NDCG@K

of precision and discounted cumulative gain. For example, Google returns search results with 66% precision and 0.47 NDCG when Top-10 results are considered. Our approach, NLP2API<sub>Google</sub>, improves the ranking and achieves a MAP@10 of 77% and a NDCG@10 of 0.61 which are 17% and 30% higher respectively. That is, although Google performs high as a *general purpose* web search engine, it might always not be precise for *code search* due to the lack of appropriate contexts. Our approach incorporates context into the search using relevant API names, and delivers more precise code search results. As shown in Table 8.6 and Fig. 8.8, similar findings were also achieved against GitHub code search and Stack Overflow native search.

**Summary of RQ<sub>5</sub>:** Our technique improves upon the result ranking of all three popular search engines using its reformulated queries. It achieves **17%** higher precision and **30%** higher NDCG than Google, i.e., the best performing search engine.

## 8.4 Threats to Validity

Threats to *internal validity* relate to experimental errors and biases. Re-implementation of the existing techniques could pose a threat. However, we used authors' implementation of RACK [201] and replicated Nie et al. [168] and Sirres et al. [229] carefully. We had multiple runs and found their best performances with

the authors’ adopted parameters which were finally chosen for comparisons. Thus, threats associated with the re-implementation might be mitigated.

Our code corpus (Section 8.3.1) contains 4,170 documents including 310 ground truth code segments. It is limited compared to a real life corpus (e.g., GitHub). However, our corpus might be sufficient enough for *comparing* a *generic NL query* with a *reformulated query* in code retrieval. Please note that our goal is to *reformulate* a query effectively for code search. Besides, we compared with three popular search engines and demonstrated the potential of our query reformulations.

Threats to *external validity* relate to generalizability of a technique. Although we experimented with Java based Q & A threads and tasks, our technique could be adapted easily for other programming languages given that code segments and API classes are extracted correctly from Stack Overflow.

## 8.5 Related Work

**Relevant API Suggestion:** There have been several studies [63, 92, 124, 138, 152, 243] that return relevant functions, API classes and methods against natural language queries. McMillan et al. [152] employ natural language processing (NLP), PageRank and spreading activation network (SAN) on a large corpus (e.g., FreeBSD), and identify functions relevant to a given query. Although they apply advanced approach for function ranking (e.g., PageRank), their candidate functions were selected using simple textual similarity which is subject to vocabulary mismatch issues [83]. On the contrary, we apply pseudo-relevance feedback, PageRank and TF-IDF for selecting the candidate API classes. Chan et al. [63] apply sophisticated graph mining techniques and return relevant API elements as a connected sub-graph. However, mining a large corpus could be very costly. Thung et al. [243] mine API documentations and feature history, and suggest relevant methods for an incoming feature request. However, this approach is project-specific and does not overcome the vocabulary mismatch issues. Rahman et al. [201] apply two heuristics derived from keyword-API co-occurrences in Stack Overflow Q & A threads, and attempt to counteract the vocabulary mismatch issues during API suggestion. Unfortunately, their approach suffers from low precision due to the adoption of simple co-occurrences. On the contrary, we (1) exploit query-API co-occurrence using a skip-gram based probabilistic model (i.e., *fastText* [54, 156]), and (2) employ pseudo-relevance feedback, Borda count and PageRank algorithm, and thus, (3) provide a novel solution that partially overcomes the limitations of earlier approaches. Rahman et al. is the most closely related work to ours in API suggestion. We compare ours with this work, and the detail comparison can be found in Section 8.3.3. Gvero and Kuncak [92] accept free-form NL queries, perform natural language processing, statistical language modelling on source code and suggest relevant method signatures. There exist other works that provide relevant code for natural language queries [44, 46, 58, 118], test cases [132, 133, 209], structural contexts [107], dependencies [254], and API class types [240, 264]. On the contrary, we collect relevant API classes for free-form NL queries by mining crowd generated knowledge stored in Stack Overflow questions and answers.

**Query Reformulation for Code Search:** Several earlier studies [92, 104, 124, 136, 138, 144, 151, 168, 251] reformulate a natural language query to improve the search for relevant code or software artefacts. Hill et al. [104] expand a natural language query by collecting frequently co-occurring terms in the method and field signatures. Conversely, we apply a different context (i.e., Q & A pairs) and a more sophisticated co-occurrence mining (e.g., skip-gram model). Lu et al. [144] expand a search query by using part of speech (POS) tagging and WordNet synonyms. Lemos et al. [134] combine WordNet and test cases in the query reformulation. However, WordNet is based on natural language corpora, and existing findings suggest that it might not be effective for synonym suggestion in software contexts [233]. On the contrary, we use a software-specific corpus (e.g., programming Q & A site), and more importantly, apply relevant API classes to query reformulation. Wang et al. [251] employ relevance feedback from developers to improve code search. Recently, Nie et al. [168] collect pseudo-relevance feedback from Stack Overflow, and reformulate a natural language query using Rocchio’s method. However, their suggested terms are natural language terms which might not be effective enough for code search given the existing evidence [45]. Another contemporary work [229] simply relies on Lucene to identify appropriate program elements from Stack Overflow answers for query reformulation. On the contrary, we employ PRF, PageRank, TF-IDF, Borda count and word semantics, and provide relevant API classes for query reformulation. The above two works are the most closely related to ours. We compare with them empirically, and the detail comparison can be found in Section 8.3.4. There exist other studies that search source code [124, 261], project repository [136], and artefact repository [138] by reformulating natural language queries. There also exist a number of query reformulation techniques [65, 84, 98, 120, 121, 188, 191, 226, 231] for concept/feature/bug/concern location. However, they suggest project-specific terms (e.g., domain terms [94]) rather than relevant API classes (like we do) for query reformulations. Hence, such terms might not be effective enough for code search on a large corpus (i.e., Internet-scale code search) that contains cross-domain projects.

In short, we meticulously bring together *crowd generated knowledge* [168], *word semantics* [54], and several IR-based approaches to effectively solve a complex Software Engineering problem, i.e., query reformulation for Internet-scale code search, which was not done by the earlier studies. Our query reformulation technique can also be employed on top of the existing code/web search engines for improving their code search performances (i.e., RQ<sub>5</sub>).

## 8.6 Summary

Software maintenance costs a significant amount of development time and efforts [88]. Developers often search for relevant code examples on the Internet [55], and reuse them in various maintenance tasks (e.g., new feature addition). As in the local code searches (e.g., concept location, bug location), developers also face major query construction challenges in the Internet-scale code search. In this chapter, we propose a novel technique—NLP2API—that accepts a programming task description as a query, reformulates the query with

relevant API classes by leveraging query-API semantic distance and by mining crowd knowledge from Stack Overflow, and then delivers an improved, reformulated query for code search on the web. Experiments using 310 queries report that our technique (1) suggests ground truth API classes with 48% precision and 58% recall for 82% of the queries, and (2) improves the given search queries significantly through reformulations. Comparisons with three state-of-the-art techniques and three popular search engines not only validate our empirical findings but also demonstrate the superiority of our technique.

In future, we plan to further investigate the potential of our skip-gram model constructed from Stack Overflow corpus. Since this model (a.k.a., word embedding technology) offers a geometric representation (e.g., vector) for word semantics, more complex semantic analyses could be performed using the geometric theories. Such analyses might (1) better explain the intent behind a given query for the code search or oppositely (2) better reveal the specification of a given code segment.

# CHAPTER 9

## CONCLUSION

### 9.1 Concluding Remarks

Software bugs and failures cost trillions of dollars every year [1, 28] and even lead to deadly accidents (e.g., Therac-25 accident<sup>1</sup>). Finding and fixing these bugs upfront consume about 50% of the development time and efforts [28, 81, 88]. While software bugs and errors are already hard to tackle, developers also receive hundreds if not thousands of change requests during software maintenance [81, 88]. Adding new features to already delivered software systems also claims about 60% of the maintenance costs [88]. Thus, resolving the bugs and addressing the change requests are two major parts of software maintenance.

The very first challenge of the two maintenance tasks above is to identify the exact locations in the source code that need to be repaired, modified or enhanced. One needs to find out the exact locations where the bug should be fixed or the existing feature that should be enhanced. Unfortunately, given million lines of code and inherent complexities in the modern software systems, identification of such locations is extremely challenging. Locating the buggy code against a bug report is called **bug localization** [276]. On the contrary, locating the target code against a change request is known as **concept location** [98, 120]. In essence, both bug localization and concept location are a special type of code search that is performed within a software system. Besides these specialized searches, developers also search for relevant code examples on the web, and reuse them in various software maintenance tasks (e.g., implementing new features). This type of code search is often called as **Internet-scale code search** [45, 151].

Every search operation above requires a query that reflects the information needs. During maintenance, software developers attempt to (1) construct search queries from the change requests for concept location, (2) construct search queries from the bug reports for bug localization, and (3) choose meaningful keywords on the fly for Internet-scale code search. Unfortunately, even the experienced developers often fail to choose the right search queries [83, 120, 125, 142]. That is, whether it is bug localization, concept location or Internet-scale code search, appropriate query construction is a major challenge. Thus, software developers are badly in need of automated tool supports for query construction during the code search.

Automated support in constructing search queries for *local code searches* (e.g., concept location, bug localization, feature location) has been an active research topic for over a decade [64, 65, 84, 95, 98, 104, 109,

---

<sup>1</sup><https://bit.ly/2KU9IR2>

120, 226, 231, 265]. There also exist a number of studies on *Internet-scale code search* [134, 135, 144, 151, 168, 251, 274] that reformulate a free-form natural language query with more appropriate keywords (e.g., relevant API classes [274]). Unfortunately, the existing literature on query reformulation is far from adequate. According to our systematic literature review, they suffer from several major limitations as follows.

First, although TF-IDF [114] has been extensively used by the existing literature [98, 120], it suffers from a major limitation. TF-IDF assumes the notion of *term independence* [137] and overlooks the semantic or syntactic dependencies among the terms during their weight calculation [53, 153]. However, such dependencies are a crucial factor in determining the term semantics or term importance [53, 157, 272]. Thus, TF-IDF might fail to deliver the appropriate search keywords from the change requests or the bug reports. As a result, TF-IDF based search queries might perform poorly in localizing the desired concepts or the bugs within the source code of a software system.

Second, regular texts and source code differ significantly from each other in their syntax, semantics and structures. While regular texts are rich in vocabulary, source code is poor in vocabulary but rich in structures or dependencies [102]. TF-IDF has been frequently used for keyword selection from the source code [84, 96, 98]. Due to term independence assumption, TF-IDF fails to capture the structural aspects of the code and simply relies on the vocabulary. Thus, it might also not be able to deliver the appropriate search keywords from the source code documents for search query reformulation.

Third, bug reports could be *noisy* containing stack traces or *poor* containing no localization hints (e.g., class names) [248]. However, existing studies [130, 167, 207, 220, 230, 249, 276] overlook such a dimension of report quality, and use almost verbatim texts from the bug report as a search query for bug localization. As a result, their search query could be either *noisy* due to excessive structured information (e.g., stack traces) or *poor* due to the lack of localization hints. Thus, existing approaches are inherently limited and might not be able to localize the software bugs when the bug reports are noisy or poor [193, 248, 276].

Fourth, search queries are often expanded with relevant program elements (e.g., API classes, methods) in Internet-scale code search [45]. Many existing studies [63, 147, 152, 271] rely on the *lexical similarity* between a given query and the API documentations of the candidate APIs for relevant API selection. Such an approach warrants that the given query should be carefully constructed, and the developer should or must possess a certain level of experience with the target APIs beforehand. Thus, the existing approaches on query reformulations might not work well if a given query (1) is not carefully constructed or (2) is not lexically similar to the documentations of a relevant API.

In this thesis, we attempt to overcome the above four challenges (1) by proposing graph-based term weighting algorithms (e.g., CodeRank [189]) that outperform TF-IDF, (2) by leveraging bug report quality dynamics and source document structures that were previously overlooked, (3) by harnessing the crowd knowledge from Stack Overflow which was previously untapped, and (4) by exploiting the semantics of the given queries and candidate keywords derived from 1.40 million Q&A threads of Stack Overflow, which was previously unexplored. Our goal was to **deliver effective solutions for source code search during**

**software maintenance with automated query reformulations**, let it be concept location, bug localization or even the Internet-scale code search. In particular, we conduct two studies (Chapters 3, 4) targeting concept location, two studies (Chapters 5, 6) targeting bug localization, and two more studies (Chapters 7, 8) targeting Internet-scale code search as follows.

- (a) The first study –**STRICT** (Chapter 3)– accepts a change request as a search query, employs graph-based term weighting algorithms, query difficulty analysis and machine learning for keyword selection from the request texts, and then delivers an improved, reformulated search query for concept location. Experiments using 2,885 change requests suggest that (1) our reformulated queries outperform the baseline search queries with a significant margin, (2) our graph-based term weighting method is a better alternative than TF-IDF for keyword selection, and (3) our approach equipped with machine learning outperforms the state-of-the-art approaches [120, 213] in constructing queries from the change requests for the concept location task.
- (b) The second study –**ACER** (Chapter 4)– accepts a poor search query as input, collects complementary keywords from the relevant source code documents by employing a graph-based term weighting method (CodeRank) and by leveraging the source document structures (e.g., method signatures, field signatures), and then delivers an improved, reformulated search query for concept location. Experiments using 1,675 queries report that our algorithm –CodeRank– that leverages the structural aspects of source code outperforms the traditional approach (e.g., TF-IDF) in keyword selection. Our query reformulation approach –ACER– also outperforms five existing studies [98, 104, 212, 213, 231] including the state-of-the-art [98] in reformulating the search queries for concept location.
- (c) The third study –**BLIZZARD** (Chapter 5)– accepts a bug report as a search query, employs appropriate methodologies for keyword selection from the report texts based on the report quality (e.g., noisy, poor), and then delivers an improved, reformulated query for bug localization. Unlike the existing studies, our approach adopts appropriate methodologies (1) to mitigate the noise from noisy bug reports and (2) to complement the poor bug reports that lack localization hints. Experiments using 5,139 bug reports suggest that (1) our reformulated queries outperform the baseline queries significantly, and (2) our approach outperforms the state-of-the-art studies on IR-based localization [220, 250, 276] and on query reformulation [191, 212, 213, 231] with significant margins.
- (d) The fourth study –**BLADER** (Chapter 6)– accepts a poor bug report as a search query, identifies appropriate candidate keywords from the relevant source code by analysing the clustering tendency between the query and the candidate keywords in terms of their underlying semantics, and then delivers an improved, reformulated query for the bug localization. Experiments using 1,546 poor bug reports suggest that (1) our reformulated queries outperform the baseline poor queries with a significant margin, and (2) our approach, **BLADER**, outperforms eleven existing studies from literature not only in IR-based bug localization [192, 220, 250, 268, 276] but also in automated search query reformulation [188, 191, 192, 212, 213, 231].

- (e) The fifth study –**RACK** (Chapter 7)– accepts a free-form query on a programming task, expands the query with relevant API classes carefully mined from Stack Overflow, and then delivers an improved, reformulated query for Internet-scale code search. Experiments using 175 free-form search queries report that (1) our reformulated queries outperform the baseline queries significantly, (2) our approach outperforms three existing studies [168, 243, 274] including the state-of-the-art [243], and (3) our reformulated queries can significantly improve the performance of three traditional web/code search engines (e.g., Google, GitHub native search, Stack Overflow native search) in the Internet-scale code search.
- (f) The sixth study –**NLP2API** (Chapter 8)– accepts a free-form query on a programming task, expands the query with relevant API classes that are selected based on query-API semantic distance analysis and crowd knowledge of Stack Overflow, and finally delivers an improved, reformulated query for Internet-scale code search. Experiments using 310 free-form search queries report that (1) our reformulated queries outperform the baseline free-form queries with a significant margin and (2) our approach outperforms three existing studies [168, 201, 229] including the state-of-the-art [229] and significantly improves three traditional web/code search engines in Internet-scale code search.

Given the above studies and their findings in our thesis, we conclude the following: (1) our graph-based term weighting approach is much more effective than the traditional alternatives (e.g., TF-IDF [98, 120]) for delivering the search keywords from the source code documents, change requests and bug reports (2) bug report quality is crucial to appropriate query construction for the bug localization, (3) source document structures can offer multiple feasible options to reformulate a given query, and (4) crowd knowledge and word semantics derived from Stack Overflow Q&A threads could be the effective means for mitigating the vocabulary mismatch problems both in local and Internet-scale code searches. Since our proposed approaches embody these solutions, they have a high potential for improving search queries and thus code searches in the Software Engineering contexts. Furthermore, each of our conducted studies could be replicated using our publicly available replication packages (Appendix A).

## 9.2 Future Work

In this thesis, we deal with different challenges concerning search query reformulations in three different code search contexts – *concept location*, *bug localization* and *Internet-scale code search*. My PhD works have produced a total of 21 peer-reviewed publications. Despite these significant number of studies, we believe that there is still room for further works and many novel dimensions (inspired by this thesis) are yet to be explored. Based on our experiments, empirical analysis, and qualitative analysis, we present a list of future research directions on automated query reformulation, software debugging and code search as follows:



### 9.2.1 Promises of Keyword Selection Algorithms in IR-Based Bug Localization

Information Retrieval (IR) has been extensively used in at least 20 Software Engineering tasks including bug localization [98]. It should be noted that bug localization is a form of local code search where bug reports are assumed as search queries. A few recent studies [123, 126, 248] have pointed out the potential biases and limitations of IR-based bug localization. According to them, IR-based localization is only good when the bug reports contain localization hints (e.g., program entity names). However, they use the whole texts from a bug report as a search query and overlook the potential of optimal search queries. Mills et al. [159] recently conduct a large-scale empirical study using Genetic Algorithms and present positive evidence for bug reports and IR-based localization. They suggest that bug reports often contain sufficient keywords which could return the buggy source documents at the top-most positions of the result list. In particular, IR-based localization could succeed 67%–88% of the time even if the bug reports do not contain any localization hints. Thus, the real challenge is to automatically extract the appropriate search keywords from a given bug report for bug localization. Term weighting algorithms could play a major role in identifying such keywords. To date, existing literature adopts two types of term weighting algorithms—*frequency-based* [98, 120, 212, 213, 231, 232] and *graph-based* [187, 189, 191, 192]—for keyword selection in Software Engineering. My PhD thesis has inspired the graph-based term weighting paradigm, and demonstrated that it is a better choice than the existing alternatives (e.g., TF-IDF) for keyword selection from the bug reports. However, our in-depth investigations suggest that the literature might yet not be sufficient enough to always deliver the important keywords from a bug report. Thus, more sophisticated and efficient term weighting algorithms are warranted to improve the search queries for IR-based bug localization.

### 9.2.2 Promises of Genetic Algorithms in IR-Based Bug Localization

Mills et al. [159] demonstrate that Genetic Algorithms (GA) are capable of generating the optimal search queries from bug reports for IR-based bug localization. They use ground truth to evaluate the fitness of the candidate search queries. However, in practice, ground truth is not known beforehand during bug localization. Thus, designing an appropriate *fitness function* is a major challenge while reformulating queries with Genetic Algorithms. In particular, given two candidate search queries, the fitness function should be able to identify the better one without executing them. Several studies [96, 98, 158, 164, 189] make use of query difficulty metrics (e.g., specificity, coherency [62]) to identify the best one from a list of given queries. However, these metrics have non-linear relationships with query performance and generally work in collaboration with machine learning algorithms. Thus, they might not be an ideal choice for the fitness function. In this thesis, we use TextRank [191], POSRank [191] and WK-Core [50] as proxies to term importance, and demonstrate their superiority to the traditional alternatives (e.g., TF-IDF). Our preliminary investigation suggests that they might also have a non-linear relationship with the query performance. Thus, future works should focus on designing more appropriate fitness functions since Genetic Algorithm has the potential for delivering

the optimal search queries from the bug reports [159]. IR-based localization is not yet widely adopted by the software practitioners due to its limitations [248]. However, we believe that IR-based bug localization equipped with GA-based optimal queries could be a preferable alternative to the developers as opposed to ad hoc, costly localization processes.

### 9.2.3 Improving Term Weighting Algorithms with Useful Term Contexts

Determining importance of a term within a body of texts (e.g., bug report, source document) has long been recognized as a major challenge [114, 120]. TF-IDF is a term weighting algorithm that has been widely used both in Information Retrieval and in Software Engineering. It determines the importance of a term in isolation, and does not consider the contexts (e.g., surrounding terms) of the term. However, a term's semantics are often determined by its contexts [157, 272]. Besides, several existing studies demonstrate the benefits of incorporating contexts in the term weighting algorithms. To date, several contextual data items such as spatial code proximity [231], positional relevance [192, 232], term co-occurrences [85, 104, 189, 191, 226], syntactic dependencies [191], time-awareness [273], and structural awareness [49, 77, 192] are employed. My PhD thesis contributes to the literature by studying term co-occurrences, syntactic dependencies, and structural/hierarchical dependencies among the terms from bug reports [192], change requests [187, 191] and source code documents [189]. However, these contexts were employed by multiple studies in isolation. Future studies should investigate how combining these contextual dimensions could benefit the existing term weighting approaches (e.g., TextRank). While Genetic Algorithms can be employed to optimize their relative weights, machine learning algorithms could also be used to design even more complex, non-linear relationships between these contexts and a term's importance.

### 9.2.4 Query Worsening Minimization

Automated query reformulation comes with both benefits and costs. Existing studies suggest that automatic query reformulation might improve the search performance up to 20% [145, 273]. However, several studies also question the complete automation in query reformulation [104, 172, 228]. Automated reformulations sometimes might add noise which drifts the query away from its original topic [228]. Thus, we need such tool supports that maximize the benefits and minimize the costs of automated query reformulations. Haiduc et al. and colleagues [96, 98] first analyse the quality of search queries in the context of concept location task, and then automatically reformulate the poor queries only. A few studies [96, 98, 158, 189] including ours [182, 189] employ query difficulty metrics and machine learning to deliver the best reformulation for a given query. Despite these attempts, the risk of query worsening due to automated reformulations still remains. Like earlier studies [74, 85, 104, 251], we believe that human cognitive power could be leveraged in this case. According to Dietrich et al. [74], human developers might perform well in removing irrelevant terms from a search query, but perform poorly in adding the new relevant terms. Relevant keywords could be hidden within thousands of identifier names (e.g., classes, methods) of a system's codebase. Future works should

incorporate the strengths of both human developers and automated tools, and then minimize both (1) the cognitive burdens on the developer and (2) the costs of inappropriate query reformulations.

### 9.2.5 Improving Pseudo-Relevance Feedback (PRF)

Collecting relevance feedback on a given query from the developers could be costly and sometimes even impractical. Hence, several existing studies [98, 188, 189, 231] including ours [188, 189, 192] employ pseudo-relevance feedback (PRF) as a feasible alternative during query reformulation. That is, they naively assume the Top-K documents retrieved by a given query as *relevant*, and then suggest important keywords from them for query reformulation. These approaches have been reported to improve over the given queries [98, 145]. However, such a feedback might not help much if the given queries are already very poor [192]. Then the retrieved documents are likely to be irrelevant. While PRF has been mostly tested in local code searches (e.g., concept location, bug localization), its effectiveness in the Internet-scale code search is not well studied. During code search on the Internet, the relevance feedback results are retrieved from thousands of open source projects which could be noisy and hard to comprehend. Thus, despite a few attempts [111, 151], much investigations are yet to be done in the area of pseudo-relevance feedback. Future works should focus on designing such a relevance feedback mechanism that is cheap, light-weight, adaptive to query quality and yet reliable enough for delivering the appropriate keywords for search query reformulation.

### 9.2.6 Promises of PageRank in Term Weighting/Source Code Retrieval

In this thesis, we adapt PageRank algorithm [57] from Information Retrieval domain, and use it in search query construction for various Software Engineering tasks such as bug localization [192], concept location [187, 188, 189, 191] and Internet-scale code search [194]. Other studies [141, 152] make use of PageRank algorithm in the ranking of code examples for Internet-scale code search. PageRank operates on a graph-based structure, adopts a notion of voting/recommendation, and then identifies the most important nodes from the graph using recursive score computations (e.g., Equation 4.4) [53, 153]. Since source code is full of structures, entities, and explicit/implicit dependencies among them, it can be represented as a graph/network. Thus, unlike traditional TF-IDF, PageRank could be a better choice for keyword selection from the source code documents. Although we conduct a few studies on this topic [189, 192], further studies are warranted to better understand the true potential of PageRank algorithm in the contexts of Software Engineering (e.g., code example search, developer network analysis).

### 9.2.7 Word Embedding Technology in Query Reformulation/Code Search

Several existing studies [85, 104, 134, 144] employ English language thesauri such as WordNet [157] to expand a given query for code search. They generally expand the query with synonyms and semantically similar/relevant words. However, Sridhara et al. [233] demonstrate that the same word has two different

semantics in source code and in regular texts. Thus, English language thesaurus might be neither appropriate nor sufficient for query expansion intended for source code search. Alternatively, several studies [109, 265, 266, 272] provide software-specific thesauri (e.g., SWordNet [266]) by analysing various software repositories (e.g., source code, Stack Overflow Q&A threads). Unfortunately, construction of these thesauri is costly, and their effectiveness in the query reformulation is not yet well tested.

Recently, a few studies [194, 268, 274] including ours [194] make use of word embedding technology in determining semantic similarity/relevance between any two software specific words. They also leverage the word embeddings in reformulating queries for source code search, and report positive evidence. Word embedding technology approximates the semantics of a given word in terms of a high dimensional numeric vector. Thus, the technology reduces various text understanding tasks (e.g., synonym detection, semantic distance calculation) into simple algebraic or geometric operations. We thus believe that this technology has lots to offer not only in query reformulation but also in other text retrieval tasks of Software Engineering.

In our fourth study (Chapter 6), we go beyond semantic distance calculation with word embedding technology. We construct a large semantic hyperspace, analyse the clustering tendency between a given query and the candidate keywords in terms of their underlying semantics, and then deliver a high quality reformulated query for bug localization. In our semantic hyperspace, each embedding vector places its corresponding word as a single co-ordinate within a high dimensional semantic space. Such a convenient approximation of word semantics is likely to encourage various geometric theories into the text processing tasks of Software Engineering (e.g., observed/expected behaviour detection from bug reports [65]).

### 9.2.8 Promises of Stack Overflow in Query Reformulation/Code Search

Despite existing attempts to use code examples [47] or test cases [133, 209, 234] as search queries, developers primarily use natural language keywords as queries for code search on the web [44, 45]. Their goal is to describe a programming task with a few keywords. Unfortunately, these queries often do not work well since they lack necessary information required for the task. Existing findings [45, 135, 194, 274] suggest that inclusion of relevant API classes or methods in the query consistently improves the code search performance. Towards this goal, two of our studies [194, 206] expand a generic NL query with relevant API classes from Stack Overflow. In particular, we leverage the co-occurrences between query keywords (from *question titles*) and API classes (from *accepted answers*) in Stack Overflow threads, determine the relevance of each API to a given query (programming task) using three heuristics, and then suggest the relevant API classes for query expansion. Given our findings [194, 201, 206], we believe that Stack Overflow has lots to offer in Software Engineering. Stack Overflow could be leveraged to transform a feature request into relevant/required API classes through machine translation [139, 181]. Such API classes could then be leveraged in implementing the software feature. Since the Q&A site deals with thousands of API programming issues/bugs and corresponding code level solutions, they could be leveraged as a starting point for automatically localizing and then solving the common software bugs in the software systems.

### 9.2.9 Word Embeddings Technology for Bug Understanding/Diagnosis

Chaparro et al. [66] first extract Observed Behaviour (OB), Expected Behaviour (EB) and Steps to Reproduce (S2R) the bug from hundreds of bug reports, and identify 154 discourse patterns using Grounded Theory approach. They later make use of OB part as a reduced version of the original query (bug report) for IR-based bug localization [65] and duplicate bug report detection [67]. Identification of these components and patterns is a major step forward in the automated bug understanding/diagnosis (e.g., root cause analysis [162, 256]). While these patterns and components are extracted using Grounded Theory, they could be further investigated and possibly extended using duplicate bug reports and word embedding technology. Duplicate bug reports are likely to refer to the same or similar bugs. Thus, they are also likely to share the underlying semantics, observed behaviour, expected behaviour and even the discourse patterns. Our fourth study (Chapter 6) constructs a large semantic hyperspace using FastText [54] on the corpus of Stack Overflow. If words used in the duplicate bug reports are visualized within such a semantic hyperspace with their corresponding embedding vectors, they might provide further insights about the discourse patterns above. Since the hyperspace provides a geometric representation for word semantics, such patterns might even be explained with geometric theories. A solid understanding of such patterns could also encourage novel tools both for bug understanding/diagnosis and even for bug fixing.

### 9.2.10 Query Reformulation as a Feasible Choice for Improved Bug Localization

Antoniol et al. [37] first use Vector Space Model (VSM) in traceability link recovery. Zhou et al. [276] later use rVSM (refined VSM) and incorporate past bug reports in the IR-based bug localization. Saha et al. [220] make use of structures both from bug reports and from source code documents in localizing the bugs. Wong et al. [258] boost up bug-proneness score of a source document based on stack trace information in the bug report. Sisman and Kak [230] and Wen et al. [255] incorporate version control history in the IR-based bug localization. Finally, Wang and Lo [250] incorporate five major items – *past bug reports*, *structures*, *stack traces*, *version history* and *author history* – from the literature, and outperform the earlier approaches on IR-based localization. Thus, existing literature often adopts an incremental approach of including more and more external artifacts in the bug localization. While these artifacts have positive influences on the localization performance, their inclusion makes the proposed approaches *less scalable* and *less usable* unfortunately. Such limitations might also explain the reluctance of the practitioners in adopting IR-based localization [170, 248]. In this thesis, our studies [187, 188, 189, 191, 192] make effective use of primary resources available to the practitioners– *bug report* and *source code*, employ appropriate query reformulations, and then deliver reasonably high localization performance at low costs. Thus, we believe that query reformulation could be an important part of at least 20 IR-based SE tasks including bug localization [192], duplicate bug detection [67], bug triaging and bug report summarization. Future studies should (1) investigate the impacts of query reformulations on these tasks and (2) develop more appropriate tool supports for them.

## BIBLIOGRAPHY

- [1] Report: Software failure caused \$1.7 trillion in financial losses in 2017. URL <https://tek.io/2FBN12i>.
- [2] ACER experimental data. URL <https://goo.gl/ZkaNvd>.
- [3] Borda count. URL [https://en.wikipedia.org/wiki/Borda\\_count](https://en.wikipedia.org/wiki/Borda_count).
- [4] Software maintenance cost defined. URL <https://galorath.com/software-maintenance-costs/>.
- [5] Theoretical CDF. URL <http://stats.stackexchange.com/questions/132652>.
- [6] Debugger Source Lookup does not work with variables. URL <https://bit.ly/2xz9UQr>.
- [7] CodeJava. URL <http://www.codejava.net>.
- [8] Google custom search engine. URL <https://developers.google.com/custom-search>.
- [9] Example code snippet. URL <https://goo.gl/WSZHiC>.
- [10] Backend of GitHub search. URL <https://bit.ly/2XwakSj>.
- [11] Java2s: Java Tutorials, . URL <http://java2s.com>.
- [12] Java Language Grammar, . URL <https://github.com/antlr/grammars-v4/tree/master/java8>.
- [13] JavaDB: Java Code Examples, . URL <http://www.javadb.com>.
- [14] Jsoup: Java HTML Parser. URL <http://jsoup.org>.
- [15] KodeJava: Java Examples. URL <http://kodejava.org>.
- [16] Enterprise search: Market share. URL <https://www.datanyze.com/market-share/enterprise-search>.
- [17] A systematic literature review of automated query reformulations in source code search. URL <https://bit.ly/2JFGWUC>.
- [18] NLP2API: Replication package. URL <https://goo.gl/sJSp2D>.
- [19] Samurai prefix and suffix list. URL <https://hiper.cis.udel.edu/Samurai>.
- [20] How do i decompress a gzip file in java? URL <https://goo.gl/14QkXq>.

- [21] Reflections Library. URL <https://code.google.com/p/reflections>.
- [22] Resampling. URL <http://www.creative-wisdom.com/teaching/WBI/resampling.shtml>.
- [23] Stack Exchange Data Explorer. URL <http://data.stackexchange.com/stackoverflow>.
- [24] STRICT: Experimental Data. URL <http://homepage.usask.ca/~masud.rahman/strict>.
- [25] Stop words, 2011. URL <https://code.google.com/p/stop-words>. Accessed: June 2017.
- [26] Java keywords, 2015. URL <https://bit.ly/1Gz0V2B>.
- [27] Blizzard: Replication package, 2018. URL <https://goo.gl/NTUqcK>.
- [28] Cost of software debugging, 2019. URL <https://goo.gl/okoj21>.
- [29] BLADER–Replication Package, 2019. URL <https://goo.gl/tcVKup>.
- [30] Blizzard-experimental data, 2019. URL <https://goo.gl/toCZrs>.
- [31] Github code search, 2019. URL <https://github.com/search>.
- [32] Apache Lucene Core, 2019. URL <https://lucene.apache.org/core>.
- [33] Polygon area calculation, 2019. URL <https://goo.gl/TnXhrP>.
- [34] Polygon, 2019. URL <https://goo.gl/yVW3dR>.
- [35] Stack Exchange archive, 2019. URL <https://archive.org/download/stackexchange>.
- [36] Word2vec tutorial - the skip-gram model, 2019. URL <https://goo.gl/CixemG>.
- [37] G Antoniol, G Canfora, G Casazza, A De Lucia, and E Merlo. Recovering Traceability Links between Code and Documentation. *TSE*, 28(10):970–983, 2002.
- [38] J. Anvik, L. Hiew, and G. C. Murphy. Coping with an open bug repository. In *Proc. OOPSLA/Eclipse*, pages 35–39, 2005.
- [39] J Anvik, L Hiew, and G C Murphy. Who Should Fix This Bug? In *Proc. ICSE*, pages 361–370, 2006.
- [40] A Arif, M M Rahman, and S Y Mukta. Information Retrieval by Modified Term Weighting Method Using Random Walk Model with Query Term Position Ranking. In *Proc. ICSPS*, pages 526–530, 2009.
- [41] B Ashok, J Joy, H Liang, S K Rajamani, G Srinivasa, and V Vangala. DebugAdvisor: A Recommender System for Debugging. In *Proc. ESEC/FSE*, pages 373–382, 2009.
- [42] A Bacchelli, M Lanza, and R Robbes. Linking e-Mails and Source Code Artifacts. In *Proc. ICSE*, pages 375–384, 2010.

- [43] A Bachmann and A Bernstein. Software Process Data Quality and Characteristics: A Historical View on Open and Closed Source Projects. In *Proc. IWPSE*, pages 119–128, 2009.
- [44] S. Bajracharya, T. Ngo, E. Linstead, Y. Dou, P. Rigor, P. Baldi, and C. Lopes. Sourcerer: A search engine for open source code supporting structure-based search. In *Proc. OOPSLA-C*, pages 681–682, 2006.
- [45] S. K. Bajracharya and C. V. Lopes. Analyzing and mining a code search engine usage log. *EMSE*, 17(4-5):424–466, 2012.
- [46] S. K. Bajracharya, J. Ossher, and C. V. Lopes. Leveraging usage similarity for effective retrieval of examples in code repositories. In *Proc. FSE*, pages 157–166, 2010.
- [47] V. Balachandran. Query by example in large-scale code repositories. In *Proc. SANER*, pages 467–476, 2015.
- [48] A. Banerjee and R. N. Dave. Validating clusters using the hopkins statistic. In *Proc. FUZZ-IEEE*, volume 1, pages 149–153, 2004.
- [49] B Basset and N A Kraft. Structural Information based Term Weighting in Text Retrieval for Feature Location. In *Proc. ICPC*, pages 133–141, 2013.
- [50] V. Batagelj and M. Zaveršnik. Fast algorithms for determining (generalized) core groups in social networks. *Advances in Data Analysis and Classification*, 5(2):129–145, 2011.
- [51] N Bettenburg, S Just, A Schröter, C Weiss, R Premraj, and T Zimmermann. What Makes a Good Bug Report? In *Proc. FSE*, pages 308–318, 2008.
- [52] N Bettenburg, R Premraj, T Zimmermann, and S Kim. Extracting Structural Information from Bug Reports. In *Proc. MSR*, pages 27–30, 2008.
- [53] R Blanco and C Lioma. Graph-based Term Weighting for Information Retrieval. *Inf. Retr.*, 15(1):54–92, 2012.
- [54] P. Bojanowski, E. Grave, A. Joulin, and T. Mikolov. Enriching word vectors with subword information. *arXiv preprint arXiv:1607.04606*, 2016.
- [55] J Brandt, P J Guo, J Lewenstein, M Dontcheva, and S R Klemmer. Two Studies of Opportunistic Programming: Interleaving Web Foraging, Learning, and Writing Code. In *Proc. SIGCHI*, pages 1589–1598, 2009.
- [56] Leo Breiman. Random forests. *Mach. Learn.*, 45(1):5–32, 2001.
- [57] S Brin and L Page. The Anatomy of a Large-Scale Hypertextual Web Search Engine. *Comput. Netw. ISDN Syst.*, 30(1-7):107–117, 1998.



- [58] B. A. Campbell and C. Treude. Nlp2code: Code snippet content assist via natural language tasks. In *Proc. ICSME*, pages 628–632, 2017.
- [59] G Capobianco, A D Lucia, R Oliveto, A Panichella, and S Panichella. Improving IR-based Traceability Recovery via Noun-Based Indexing of Software Artifacts. *JSEP*, 25(7):743–762, 2013.
- [60] D Carmel and E Yom-Tov. *Estimating the Query Difficulty for Information Retrieval*. Morgan & Claypool, 2010.
- [61] D Carmel, E Yom-Tov, A Darlow, and D Pelleg. What Makes a Query Difficult? In *Proc. SIGIR*, pages 390–397, 2006.
- [62] C Carpineto and G Romano. A Survey of Automatic Query Expansion in Information Retrieval. *ACM Comput. Surv.*, 44(1):1:1–1:50, 2012.
- [63] W Chan, H Cheng, and D Lo. Searching Connected API Subgraph via Text Phrases. In *Proc. FSE*, pages 10:1—10:11, 2012.
- [64] O Chaparro and A Marcus. On the Reduction of Verbose Queries in Text Retrieval Based Software Maintenance. In *Proc. ICSE-C*, pages 716–718, 2016.
- [65] O Chaparro, J M Florez, and A Marcus. Using Observed Behavior to Reformulate Queries during Text Retrieval-based Bug Localization. In *Proc. ICSME*, pages 376–387, 2017.
- [66] O Chaparro, J Lu, F Zampetti, L Moreno, M Di Penta, A Marcus, G Bavota, and V Ng. Detecting Missing Information in Bug Descriptions. In *Proc. ESEC/FSE*, pages 396–407, 2017.
- [67] O. Chaparro, J. M. Florez, U. Singh, and A. Marcus. Reformulating queries for duplicate bug report detection. In *Proc. SANER*, page 12, 2019.
- [68] F Chen and S Kim. Crowd Debugging. In *Proc. ESEC/FSE*, pages 320–332, 2015.
- [69] B. Cleary, C. Exton, J. Buckley, and M. English. An empirical analysis of information retrieval based concept location techniques in software comprehension. *EMSE*, 14(1):93–130, 2009.
- [70] J Cordeiro, B Antunes, and P Gomes. Context-based Recommendation to Support Problem Solving in Software Development. In *Proc. RSSE*, pages 85–89, 2012.
- [71] R. F. G. Da Silva, C. K. Roy, M. M. Rahman, K. Schneider, K. Paixão, and M. Maia. Recommending comprehensive solutions for programming tasks by mining crowd knowledge. In *Proc. ICPC*, page 11, 2019.
- [72] B Dagenais and M P Robillard. Creating and Evolving Developer Documentation: Understanding the Decisions of Open Source Contributors. In *Proc. FSE*, pages 127–136, 2010.

- [73] B Dagenais and M P Robillard. Recovering Traceability Links between an API and its Learning Resources. In *Proc. ICSE*, pages 47–57, 2012.
- [74] T. Dietrich, J. Cleland-Huang, and Y. Shin. Learning effective query transformations for enhanced requirements trace retrieval. In *Proc. ASE*, pages 586–591, 2013.
- [75] B Dit, L Guerrouj, D Poshyvanyk, and G Antoniol. Can Better Identifier Splitting Techniques Help Feature Location? In *Proc. ICPC*, pages 11–20, 2011.
- [76] M. Duijn, A. Kucera, and A. Bacchelli. Quality questions need quality code: Classifying code fragments on stack overflow. In *Proc. MSR*, pages 410–413, 2015.
- [77] Brian P. Eddy, Nicholas A. Kraft, and Jeff Gray. Impact of structural weighting on a latent dirichlet allocation based feature location technique. *JSEP*, 30(1):e1892, 2018.
- [78] F. Ensan, E. Bagheri, and M. Kahani. The application of users’ collective experience for crafting suitable search engine query recommendations. In *Proc. CNSR*, pages 148–156, 2007.
- [79] E Enslin, E Hill, L Pollock, and K Vijay-Shanker. Mining Source Code to Automatically Split Identifiers for Software Analysis. In *Proc. MSR*, pages 71–80, 2009.
- [80] S Ercan, Q Stokkink, and A Bacchelli. Automatic Assessments of Code Explanations: Predicting Answering Times on Stack Overflow. In *Proc. MSR*, pages 442–445, 2015.
- [81] L Favre. Modernizing Software & System Engineering Processes. In *Proc. ICSENG*, pages 442–447, 2008.
- [82] G. Fischer and H. Nieper-Lemke. Helgon: Extending the retrieval by reformulation paradigm. In *Proc. CHI*, pages 357–362, 1989.
- [83] G W Furnas, T K Landauer, L M Gomez, and S T Dumais. The Vocabulary Problem in Human-system Communication. *Commun. ACM*, 30(11):964–971, 1987.
- [84] G Gay, S Haiduc, A Marcus, and T Menzies. On the Use of Relevance Feedback in IR-based Concept Location. In *Proc. ICSM*, pages 351–360, 2009.
- [85] X. Ge, D. C. Shepherd, K. Damevski, and E. Murphy-Hill. Design and evaluation of a multi-recommendation system for local code search. *Journal of Visual Languages and Computing*, 39:1 – 9, 2017.
- [86] M. Ghafari and H. Moradi. A framework for classifying and comparing source code recommendation systems. In *Proc. SANER*, pages 555–556, 2017.
- [87] M. Gibiec, A. Czauderna, and J. Cleland-Huang. Towards mining replacement queries for hard-to-retrieve traces. In *Proc. ASE*, pages 245–254, 2010.

- [88] R. L. Glass. Frequently forgotten fundamental facts about software engineering. *IEEE Software*, 18(3):112–111, 2001.
- [89] J Gosling, B Joy, G Steele, and G Bracha. The Java Language Specification: Java SE 7 Edition. 2012.
- [90] X. Gu, H. Zhang, and S. Kim. Deep code search. In *Proc. ICSE*, pages 933–944, 2018.
- [91] Z Gu, E T Barr, D Schleck, and Z Su. Reusing Debugging Knowledge via Trace-based Bug Search. In *Proc. OOPSLA*, pages 927–942, 2012.
- [92] T Gvero and V Kuncak. Interactive Synthesis Using Free-form Queries. In *Proc. ICSE*, pages 689–692, 2015.
- [93] S Haiduc. Automatically Detecting the Quality of the Query and its Implications in IR-based Concept Location. In *Proc. ASE*, pages 637–640, 2011.
- [94] S. Haiduc and A. Marcus. On the Use of Domain Terms in Source Code. In *Proc. ICPC*, pages 113–122, 2008.
- [95] S Haiduc and A Marcus. On the Effect of the Query in IR-based Concept Location. In *Proc. ICPC*, pages 234–237, jun 2011.
- [96] S Haiduc, G Bavota, R Oliveto, A De Lucia, and A Marcus. Automatic Query Performance Assessment During the Retrieval of Software Artifacts. In *Proc. ASE*, pages 90–99, 2012.
- [97] S Haiduc, G Bavota, R Oliveto, A Marcus, and A De Lucia. Evaluating the Specificity of Text Retrieval Queries to Support Software Engineering Tasks. In *Proc. ICSE*, pages 1273–1276, 2012.
- [98] S Haiduc, G Bavota, A Marcus, R Oliveto, A De Lucia, and T Menzies. Automatic Query Reformulations for Text Retrieval in Software Engineering. In *Proc. ICSE*, pages 842–851, 2013.
- [99] S Haiduc, G De Rosa, G Bavota, R Oliveto, A De Lucia, and A Marcus. Query Quality Prediction and Reformulation for Source Code Search: the Refoqus Tool. In *Proc. ICSE*, pages 1307–1310, 2013.
- [100] Z Harris. Mathematical Structures in Language Contents. 1968.
- [101] S Hassan, R Mihalcea, and C Banea. Random-Walk Term Weighting for Improved Text Classification. In *Proc. ICSC*, pages 242–249, 2007.
- [102] V. J. Hellendoorn and P. Devanbu. Are deep neural networks the best choice for modeling source code? In *Proc. ESEC/FSE*, pages 763–773, 2017.
- [103] K. Herzig and A. Zeller. The impact of tangled code changes. In *Proc. MSR*, pages 121–130, 2013.
- [104] E Hill, L Pollock, and K Vijay-Shanker. Automatically Capturing Source Code Context of NL-queries for Software Maintenance and Reuse. In *Proc. ICSE*, pages 232–242, 2009.

- [105] E Hill, L Pollock, and K Vijay-Shanker. Improving Source Code Search with Natural Language Phrasal Representations of Method Signatures. In *Proc. ASE*, pages 524–527, 2011.
- [106] E Hill, S Rao, and A Kak. On the Use of Stemming for Concern Location and Bug Localization in Java. In *Proc. SCAM*, pages 184–193, 2012.
- [107] R Holmes and G C Murphy. Using Structural Context to Recommend Source Code Examples. In *Proc. ICSE*, pages 117–125, 2005.
- [108] B. Hopkins and J. G. Skellam. A new method for determining the type of distribution of plant individuals. *Annals of Botany*, 18(70):213–227, 1954.
- [109] M J Howard, S Gupta, L Pollock, and K Vijay-Shanker. Automatically Mining Software-based, Semantically-Similar Words from Comment-Code Mappings. In *Proc. MSR*, pages 377–386, 2013.
- [110] Q. Huang, Y. Yang, X. Wang, H. Wan, R. Wang, and G. Wu. Query expansion via intent predicting. *IJSEKE*, 27(09n10):1591–1601, 2017.
- [111] Q. Huang, Y. Yang, X. Zhan, H. Wan, and G. Wu. Query expansion based on statistical learning from code changes. *SPE*, 48(7):1333–1351, 2018.
- [112] K. Järvelin and J. Kekäläinen. Cumulated gain-based evaluation of ir techniques. *ACM Trans. Inf. Syst.*, 20(4):422–446, 2002.
- [113] Otto Jespersen. *The Philosophy of Grammar*. 1929.
- [114] K. S. Jones. A statistical interpretation of term specificity and its application in retrieval. *J. Doc.*, 28(1):11–21, 1972.
- [115] Armand Joulin, Edouard Grave, Piotr Bojanowski, and Tomas Mikolov. Bag of tricks for efficient text classification. *arXiv preprint arXiv:1607.01759*, 2016.
- [116] T Kaneishi and T Dohi. Parametric Bootstrapping for Assessing Software Reliability Measures. In *Proc. PRDC*, pages 1–9, 2011.
- [117] I. Keivanloo and J. Rilling. Internet-scale java source code data set, 2011. URL <http://aseg.cs.concordia.ca/codesearch/#IJaDataSet>.
- [118] I. Keivanloo, J. Rilling, and Y. Zou. Spotting working code examples. In *Proc. ICSE*, pages 664–675, 2014.
- [119] D. Kelly and J. Teevan. Implicit feedback for inferring user preference: A bibliography. *SIGIR Forum*, 37(2):18–28, 2003.

- [120] K Kevic and T Fritz. Automatic Search Term Identification for Change Tasks. In *Proc. ICSE*, pages 468–471, 2014.
- [121] K Kevic and T Fritz. A Dictionary to Translate Change Tasks to Source Code. In *Proc. MSR*, pages 320–323, 2014.
- [122] D Kim, Y Tao, S Kim, and A Zeller. Where Should We Fix This Bug? A Two-Phase Recommendation Model. *TSE*, 39(11):1597–1610, 2013.
- [123] M. Kim and E. Lee. Are information retrieval-based bug localization techniques trustworthy? In *Proc. ICSE*, pages 248–249, 2018.
- [124] M. Kimmig, M. Monperrus, and M. Mezini. Querying source code with natural language. In *Proc. ASE*, pages 376–379, 2011.
- [125] Andrew J. Ko, Brad A. Myers, Michael J. Coblenz, and Htet Htet Aung. An exploratory study of how developers seek, relate, and collect relevant information during software maintenance tasks. *TSE*, 32(12):971–987, 2006.
- [126] P. S. Kochhar, Y. Tian, and D. Lo. Potential biases in bug localization: Do they matter? In *Proc. ASE*, pages 803–814, 2014.
- [127] G. Kumaran and V. R. Carvalho. Reducing long queries using query quality predictors. In *Proc. SIGIR*, pages 564–571, 2009.
- [128] A. N. Lam, A. T. Nguyen, H. A. Nguyen, and T. N. Nguyen. Bug localization with combination of deep learning and information retrieval. In *Proc. ICPC*, pages 218–229, 2017.
- [129] R. Lapeña, J. Font, F. Pérez, and C. Cetina. Improving feature location by transforming the query from natural language into requirements. In *Proc. SPLC*, pages 362–369, 2016.
- [130] Tien-Duy B Le, R J Oentaryo, and D Lo. Information Retrieval and Spectrum Based Bug Localization: Better Together. In *Proc. ESEC/FSE*, pages 579–590, 2015.
- [131] Vu Le, Sumit Gulwani, and Zhendong Su. Smartsynth: Synthesizing smartphone automation scripts from natural language. In *Proc. MobiSys*, pages 193–206, 2013.
- [132] O. A. L. Lemos, S. K. Bajracharya, J. Ossher, R. S. Morla, P. C. Masiero, P. Baldi, and C. V. Lopes. Codegenie: Using test-cases to search and reuse source code. In *Proc. ASE*, pages 525–526, 2007.
- [133] O. A. L. Lemos, S. Bajracharya, J. Ossher, P. C. Masiero, and C. Lopes. A test-driven approach to code search and its application to the reuse of auxiliary functionality. *IST*, 53(4):294 – 306, 2011.
- [134] O. A. L. Lemos, A. C. de Paula, F. C. Zanichelli, and C. V. Lopes. Thesaurus-based automatic query expansion for interface-driven code search. In *Proc. MSR*, pages 212–221, 2014.

- [135] O. A. L. Lemos, A. C. de Paula, H. Sajnani, and C. V. Lopes. Can the use of types and query expansion help improve large-scale code search? In *Proc. SCAM*, pages 41–50, 2015.
- [136] Z. Li, T. Wang, Y. Zhang, Y. Zhan, and G. Yin. Query reformulation by leveraging crowd wisdom for scenario-based software search. In *Proc. Internetware*, pages 36–44, 2016.
- [137] J. Lin and G. C. Murray. Assessing the term independence assumption in blind relevance feedback. In *Proc. SIGIR*, pages 635–636, 2005.
- [138] J. Lin, Y. Liu, J. Guo, J. Cleland-Huang, W. Goss, W. Liu, S. Lohar, N. Monaikul, and A. Rasin. Tiqu: A natural language interface for querying software project data. In *Proc. ASE*, pages 973–977, 2017.
- [139] Z. Lin, Y. Zou, J. Zhao, and B. Xie. Improving software text retrieval using conceptual knowledge in source code. In *Proc. ASE*, pages 123–134, 2017.
- [140] M Linares-Vásquez, G Bavota, M Di Penta, R Oliveto, and D Poshyvanyk. How Do API Changes Trigger Stack Overflow Discussions? A Study on the Android SDK. In *Proc. ICPC*, pages 83–94, 2014.
- [141] E. Linstead, S. Bajracharya, T. Ngo, P. Rigor, C. Lopes, and P. Baldi. Sourcerer: mining and searching internet-scale software repositories. *Data Mining and Knowledge Discovery*, 18(2):300–336, 2009.
- [142] D Liu, A Marcus, D Poshyvanyk, and V Rajlich. Feature Location via Information Retrieval Based Filtering of a Single Scenario Execution Trace. In *Proc. ASE*, pages 234–243, 2007.
- [143] C. Lopes, S. Bajracharya, J. Ossher, and P. Baldi. UCI source code data sets, 2010. URL <http://www.ics.uci.edu/~lopes/datasets/>.
- [144] Meili Lu, X. Sun, S. Wang, D. Lo, and Yucong Duan. Query expansion via wordnet for effective code search. In *Proc. SANER*, pages 545–549, 2015.
- [145] X Allan Lu and Robert B Keefer. Query expansion / reduction and its impact on retrieval effectiveness. In *Proc. TREC*, pages 1–9, 1995.
- [146] A. D. Lucia, R. Oliveto, and P. Sgueglia. Incremental approach and user feedbacks: a silver bullet for traceability recovery. In *Proc. ICSM*, pages 299–309, 2006.
- [147] F. Lv, H. Zhang, J. Lou, S. Wang, D. Zhang, and J. Zhao. Codehow: Effective code search based on api understanding and extended boolean model. In *Proc. ASE*, pages 260–270, 2015.
- [148] L Mamykina, B Manoim, M Mittal, G Hripcsak, and B Hartmann. Design Lessons from the Fastest Q & A Site in the West. In *Proc. CHI*, pages 2857–2866, 2011.
- [149] A Marcus and S Haiduc. Text Retrieval Approaches for Concept Location in Source Code. In *Software Engineering*, volume 7171, pages 126–158. 2013.

- [150] A Marcus, A Sergeyev, V Rajlich, and J I Maletic. An Information Retrieval Approach to Concept Location in Source Code. In *Proc. WCRE*, pages 214–223, 2004.
- [151] L. Martie, T. D. LaToza, and A. v. d. Hoek. Codeexchange: Supporting reformulation of internet-scale code queries in context (t). In *Proc. ASE*, pages 24–35, 2015.
- [152] C McMillan, M Grechanik, D Poshyvanyk, Q Xie, and C Fu. Portfolio: Finding Relevant Functions and their Usage. In *Proc. ICSE*, pages 111–120, 2011.
- [153] R Mihalcea and P Tarau. TextRank: Bringing Order into Texts. In *Proc. EMNLP*, pages 404–411, 2004.
- [154] R Mihalcea, P Tarau, and E Figa. PageRank on Semantic Networks, with Application to Word Sense Disambiguation. In *Proc. COLING*, 2004.
- [155] T. Mikolov, K. Chen, G. Corrado, and J. Dean. Efficient estimation of word representations in vector space. *CoRR*, abs/1301.3781, 2013.
- [156] T Mikolov, I Sutskever, K Chen, G S Corrado, and J Dean. Distributed Representations of Words and Phrases and their Compositionality. In *Proc. NIPS*, pages 3111–3119, 2013.
- [157] George A. Miller. Wordnet: A lexical database for english. *Commun. ACM*, 38(11):39–41, 1995.
- [158] C. Mills, G. Bavota, S. Haiduc, R. Oliveto, A. Marcus, and A. D. Lucia. Predicting query quality for applications of text retrieval to software engineering tasks. *TOSEM*, 26(1):3:1–3:45, 2017.
- [159] C. Mills, J. Pantiuchina, E. Parra, G. Bavota, and S. Haiduc. Are bug reports enough for text retrieval-based bug localization? In *Proc. ICSME*, pages 381–392, 2018.
- [160] A. Mondal, M. M. Rahman, and C. K. Roy. Embedded Emotion-based Classification of Stack Overflow Questions Towards the Question Quality Prediction. In *Proc. SEKE*, pages 521–526, 2016.
- [161] S. Mondal, M. M. Rahman, and C. K. Roy. Can issues reported at stack overflow questions be reproduced? an exploratory study. In *Proc. MSR*, page 11, 2019.
- [162] K. Moran, M. Linares-Vásquez, C. Bernal-Cárdenas, C. Vendome, and D. Poshyvanyk. Automatically discovering, reporting and reproducing android application crashes. In *Proc. ICST*, pages 33–44, 2016.
- [163] L. Moreno, J. J. Treadway, A. Marcus, and W. Shen. On the use of stack traces to improve text retrieval-based bug localization. In *Proc. ICSME*, pages 151–160, 2014.
- [164] L Moreno, G Bavota, S Haiduc, M Di Penta, R Oliveto, B Russo, and A Marcus. Query-based Configuration of Text Retrieval Solutions for Software Engineering Tasks. In *Proc. ESEC/FSE*, pages 567–578, 2015.

- [165] D Mujumdar, M Kallenbach, B Liu, and B Hartmann. Crowdsourcing Suggestions to Programming Problems for Dynamic Web Development Languages. In *Proc. CHI*, pages 1525–1530, 2011.
- [166] K. Nakasai, M. Tsunoda, and H. Hata. Web search behaviors for software development. In *Proc. CHASE*, pages 125–128, 2016.
- [167] A T Nguyen, T T Nguyen, J Al-Kofahi, H V Nguyen, and T N Nguyen. A Topic-based Approach for Narrowing the Search Space of Buggy Files from a Bug Report. In *Proc. ASE*, pages 263–272, 2011.
- [168] L. Nie, H. Jiang, Z. Ren, Z. Sun, and X. Li. Query expansion based on crowd knowledge for code search. *TSC*, 9(5):771–783, 2016.
- [169] F. J. Ortega, C. Macdonald, J. A. Troyano, and F. Cruz. Spam detection with a content-based random-walk algorithm. In *Proc. SMUC*, pages 45–52, 2010.
- [170] C Parnin and A Orso. Are Automated Debugging Techniques Actually Helping Programmers? In *Proc. ISSTA*, pages 199–209, 2011.
- [171] C Parnin and C Treude. Measuring API Documentation on the Web. In *Proc. Web2SE*, pages 25–30, 2011.
- [172] F. Perez, J. Font, L. Arcega, and C. Cetina. Automatic query reformulations for feature location in a model-based family of software products. *Data & Knowledge Engineering*, 116:159 – 176, 2018.
- [173] N Pingclasai, H Hata, and K i. Matsumoto. Classifying Bug Reports to Bugs and Other Requests Using Topic Modeling. In *Proc. APSEC*, volume 2, pages 13–18, 2013.
- [174] R. Polikar. Ensemble based systems in decision making. *Proc. MCAS*, 6(3):21–45, 2006.
- [175] L Ponzanelli, A Bacchelli, and M Lanza. Seahawk: Stack Overflow in the IDE. In *Proc. ICSE*, pages 1295–1298, 2013.
- [176] L Ponzanelli, G Bavota, M D Penta, R Oliveto, and M Lanza. Prompter: A Self-Confident Recommender System. In *Proc. ICSME*, pages 577–580, 2014.
- [177] L. Ponzanelli, A. Mocci, A. Bacchelli, M. Lanza, and D. Fullerton. Improving Low Quality Stack Overflow Post Detection. In *Proc. ICSME*, pages 541–544, 2014.
- [178] D Poshyvanyk and A Marcus. Combining Formal Concept Analysis with Information Retrieval for Concept Location in Source Code. In *Proc. ICPC*, pages 37–48, 2007.
- [179] D Poshyvanyk, Y G Gueheneuc, A Marcus, G Antoniol, and V Rajlich. Feature Location Using Probabilistic Ranking of Methods Based on Execution Scenarios and Information Retrieval. *TSE*, 33(6):420–432, 2007.



- [180] R. Quinlan. *C4.5: Programs for Machine Learning*. Morgan Kaufmann Publishers, 1993.
- [181] M. Raghthaman, Y. Wei, and Y. Hamadi. Swim: Synthesizing what i mean: Code search and idiomatic snippet synthesis. In *Proc. ICSE*, pages 357–367, 2016.
- [182] M. M. Rahman. Supporting code search with context-aware, analytics-driven, effective query reformulation. In *Proc. ICSE-C*, page 4, 2019.
- [183] M. M. Rahman and C. K. Roy. Surfclipse: Context-aware meta-search in the ide. In *Proc. ICSME*, pages 617–620, 2014.
- [184] M. M. Rahman and C. K. Roy. On the use of context in recommending exception handling code examples. In *Proc. SCAM*, pages 285–294, 2014.
- [185] M. M. Rahman and C. K. Roy. Recommending relevant sections from a webpage about programming errors and exceptions. In *Proc. CASCAN*, pages 181–190, 2015.
- [186] M. M. Rahman and C. K. Roy. An insight into the unresolved questions at stack overflow. In *Proc. MSR*, pages 426–429, 2015.
- [187] M M Rahman and C K Roy. TextRank Based Search Term Identification for Software Change Tasks. In *Proc. SANER*, pages 540–544, 2015.
- [188] M M Rahman and C K Roy. QUICKAR: Automatic Query Reformulation for Concept Location Using Crowdsourced Knowledge. In *Proc. ASE*, pages 220–225, 2016.
- [189] M M Rahman and C K Roy. Improved Query Reformulation for Concept Location using CodeRank and Document Structures. In *Proc. ASE*, pages 428–439, 2017.
- [190] M. M. Rahman and C. K. Roy. Impact of continuous integration on code reviews. In *Proc. MSR*, pages 499–502, 2017.
- [191] M M Rahman and C K Roy. STRICT: Information Retrieval Based Search Term Identification for Concept Location. In *Proc. SANER*, pages 79–90, 2017.
- [192] M. M. Rahman and C. K. Roy. Improving ir-based bug localization with context-aware query reformulation. In *Proc. ESEC/FSE*, pages 621–632, 2018.
- [193] M. M. Rahman and C. K. Roy. Improving bug localization with report quality dynamics and query reformulation. In *Proc. ICSE-C*, pages 348–349, 2018.
- [194] M. M. Rahman and C. K. Roy. Effective reformulation of query for code search using crowdsourced knowledge and extra-large data analytics. In *Proc. ICSME*, pages 516–527, 2018.

- [195] M. M. Rahman and C. K. Roy. Nlp2api: Query reformulation for code search using crowdsourced knowledge and extra-large data analytics. In *Proc. ICSME*, page 714, 2018.
- [196] M. M. Rahman and Chanchal K. Roy. An insight into the pull requests of github. In *Proc. MSR*, pages 364–367.
- [197] M. M. Rahman, S. Yeasmin, and C. K. Roy. An ide-based context-aware meta search engine. In *Proc. WCRE*, pages 467–471, 2013.
- [198] M M Rahman, S Yeasmin, and C K Roy. Towards a Context-Aware IDE-Based Meta Search Engine for Recommendation about Programming Errors and Exceptions. In *Proc. CSMR-WCRE*, pages 194–203, 2014.
- [199] M. M. Rahman, C. K. Roy, and I. Keivanloo. Recommending Insightful Comments for Source Code using Crowdsourced Knowledge. In *Proc. SCAM*, pages 81–90, 2015.
- [200] M M Rahman, C K Roy, and J Collins. CORRECT: Code Reviewer Recommendation Based on Cross-Project and Technology Experience. In *Proc. ICSE*, page to appear, 2016.
- [201] M M Rahman, C K Roy, and D Lo. RACK: Automatic API Recommendation using Crowdsourced Knowledge. In *Proc. SANER*, pages 349–359, 2016.
- [202] M. M. Rahman, C. K. Roy, J Redl, and J. Collins. CORRECT: Code Reviewer Recommendation at GitHub for Vendasta Technologies. In *Proc. ASE*, pages 792–797, 2016.
- [203] M. M. Rahman, C. K. Roy, and R. G. Kula. Predicting usefulness of code review comments using textual features and developer experience. In *Proc. MSR*, pages 215–226, 2017.
- [204] M. M. Rahman, C. K. Roy, and D. Lo. Rack: Code search in the ide using crowdsourced knowledge. In *Proc. ICSE-C*, pages 51–54, 2017.
- [205] M. M. Rahman, J. Barson, S. Paul, J. Kayani, F. A. Lois, S. F. Quezada, C. Parnin, K T. Stolee, and Baishakhi Ray. Evaluating how developers use general-purpose web-search for code retrieval. In *Proc. MSR*, page 10, 2018.
- [206] M. M. Rahman, C. K. Roy, and D. Lo. Automatic query reformulation for code search using crowdsourced knowledge. *EMSE*, page 56, 2018.
- [207] S Rao and A Kak. Retrieval from Software Libraries for Bug Localization: A Comparative Study of Generic and Composite Text Models. In *Proc. MSR*, pages 43–52, 2011.
- [208] S Rastkar, G C Murphy, and G Murray. Summarizing Software Artifacts: A Case Study of Bug Reports. In *Proc. ICSE*, pages 505–514, 2010.

- [209] Steven P. Reiss. Semantics-based code search. In *Proc. ICSE*, pages 243–253, 2009.
- [210] M. Revelle, B. Dit, and D. Poshyvanyk. Using data fusion and web mining to support feature location in software. In *Proc. ICPC*, pages 14–23, 2010.
- [211] P C Rigby and M P Robillard. Discovering Essential Code Elements in Informal Documentation. In *Proc. ICSE*, pages 832–841, 2013.
- [212] S. E. Robertson. On term selection for query expansion. *J. Doc.*, 46(4):359–364, 1991.
- [213] J J Rocchio. *The SMART Retrieval System—Experiments in Automatic Document Processing*. Prentice-Hall, Inc.
- [214] Lior Rokach. Ensemble-based classifiers. *JAIR*, 33(1):1–39, 2010.
- [215] M. Roldan-Vega, G. Mallet, E. Hill, and J. A. Fails. Conquer: A tool for nl-based query refinement and contextualizing code search results. In *Proc. ICSM*, pages 512–515, 2013.
- [216] J. Romano, J.D. Kromrey, J. Coraggio, and J. Skowronek. Appropriate statistics for ordinal level data: Should we really be using t-test and Cohen’sd for evaluating group differences on the NSSE and other surveys? In *Annual meeting of the Florida Association of Institutional Research*, pages 1–3, 2006.
- [217] F. Rousseau and M. Vazirgiannis. Main core retention on graph-of-words for single-document keyword extraction. In *Proc. ECIR*, pages 382–393, 2015.
- [218] C K Roy and J R Cordy. NICAD: Accurate Detection of Near-Miss Intentional Clones Using Flexible Pretty-Printing and Code Normalization. In *Proc. ICPC*, pages 172–181, 2008.
- [219] C. Sadowski, K. T. Stolee, and S. Elbaum. How developers search for code: A case study. In *Proc. ESEC/FSE*, pages 191–201, 2015.
- [220] R K Saha, M Lease, S Khurshid, and D E Perry. Improving Bug Localization using Structured Information Retrieval. In *Proc. ASE*, pages 345–355, 2013.
- [221] R. K. Saha, J. Lawall, S. Khurshid, and D. E. Perry. On the effectiveness of information retrieval based bug localization for c programs. In *Proc. ICSME*, pages 161–170, 2014.
- [222] G. Salton and C. Buckley. Readings in information retrieval. chapter Improving Retrieval Performance by Relevance Feedback, pages 355–364. 1997.
- [223] A. Satter and K. Sakib. A search log mining based query expansion technique to improve effectiveness in code search. In *Proc. ICCIT*, pages 586–591, 2016.
- [224] T Savage, M Revelle, and D Poshyvanyk. FLAT3: Feature Location and Textual Tracing Tool. In *Proc. ICSE*, pages 255–258, 2010.

- [225] G Scanniello and A Marcus. Clustering Support for Static Concept Location in Source Code. In *Proc. ICPC*, pages 1–10, 2011.
- [226] D Shepherd, Z P Fry, E Hill, L Pollock, and K Vijay-Shanker. Using Natural Language Program Analysis to Locate and Understand Action-Oriented Concerns. In *Proc. ASOD*, pages 212–224, 2007.
- [227] Z Shi, J Keung, and Q Song. An Empirical Study of BM25 and BM25F Based Feature Location Techniques. In *Proc. InnoSWDev*, pages 106–114, 2014.
- [228] A. Shtok, O. Kurland, D. Carmel, F. Raiber, and G. Markovits. Predicting query performance by query-drift estimation. *TOIS*, 30(2):11:1–11:35, 2012.
- [229] R. Sirres, T. F. Bissyandé, D. Kim, D. Lo, J. Klein, K. Kim, and Y. L. Traon. Augmenting and structuring user queries to support efficient free-form code search. *EMSE*, pages 2622–2654, 2018.
- [230] B Sisman and A C Kak. Incorporating Version Histories in Information Retrieval Based Bug Localization. In *Proc. MSR*, pages 50–59, 2012.
- [231] B Sisman and A C Kak. Assisting Code Search with Automatic Query Reformulation for Bug Localization. In *Proc. MSR*, pages 309–318, 2013.
- [232] B. Sisman, S. A. Akbar, and A. C. Kak. Exploiting spatial code proximity and order for improved source code retrieval for bug localization. *JSEP*, 29(1):e1805, 2017.
- [233] G Sridhara, E Hill, L Pollock, and K Vijay-Shanker. Identifying Word Relations in Software: A Comparative Study of Semantic Similarity Tools. In *Proc. ICPC*, pages 123–132, 2008.
- [234] K. T. Stolee, S. Elbaum, and M. B. Dwyer. Code search with input/output queries: Generalizing, ranking, and assessment. *JSS*, 116(C):35–48, 2016.
- [235] J. Svajlenko and C. K. Roy. Fast, scalable and user-guided clone detection. In *Proc. ICSE-C*, pages 352–353, 2018.
- [236] J. Svajlenko, J. F. Islam, I. Keivanloo, C. K. Roy, and M. M. Mia. Towards a big data curated benchmark of inter-project code clones. In *Proc. ICSME*, pages 476–480, 2014.
- [237] M Tan, L Tan, S Dara, and C Mayeux. Online Defect Prediction for Imbalanced Data. In *Proc. ICSE*, volume 2, pages 99–108, 2015.
- [238] D. Tang, F. Wei, N. Yang, M. Zhou, T. Liu, and B. Qin. Learning sentiment-specific word embedding for twitter sentiment classification. In *Proc. ACL*, pages 1555–1565, 2014.
- [239] P Thongtanunam, R G Kula, N Yoshida, H Iida, and K Matsumoto. Who Should Review My Code? In *Proc. SANER*, pages 141–150, 2015.

- [240] S Thummalapenta and T Xie. Parseweb: A Programmer Assistant for Reusing Open Source Code on the Web. In *Proc. ASE*, pages 204–213, 2007.
- [241] F Thung, D Lo, and L Jiang. Automatic Defect Categorization. In *Proc. WCRE*, pages 205–214, 2012.
- [242] F Thung, D Lo, and J Lawall. Automated Library Recommendation. In *Proc. WCRE*, pages 182–191, 2013.
- [243] F Thung, S Wang, D Lo, and J Lawall. Automatic Recommendation of API Methods from Feature Requests. In *Proc. ASE*, pages 290–300, 2013.
- [244] K Toutanova, D Klein, C D Manning, and Y Singer. Feature-Rich Part-of-Speech Tagging with a Cyclic Dependency Network. In *Proc. HLT-NAACL*, pages 252–259, 2003.
- [245] Y. Uneno, O. Mizuno, and E. H. Choi. Using a distributed representation of words in localizing relevant files for bug reports. In *Proc. QRS*, pages 183–190, 2016.
- [246] C Vassallo, S Panichella, M Di Penta, and G Canfora. CODES: Mining Source Code Descriptions from Developers Discussions. In *Proc. ICPC*, pages 106–109, 2014.
- [247] I Vessey. Expertise in Debugging Computer Programs: An Analysis of the Content of Verbal Protocols. *TSMC*, 16(5):621–637, 1986.
- [248] Q Wang, C Parnin, and A Orso. Evaluating the Usefulness of IR-based Fault Localization Techniques. In *Proc. ISSTA*, pages 1–11, 2015.
- [249] S Wang and D Lo. Version History, Similar Report, and Structure: Putting Them Together for Improved Bug Localization. In *Proc. ICPC*, pages 53–63, 2014.
- [250] S. Wang and D. Lo. Amalgam+: Composing rich information sources for accurate bug localization. *JSEP*, 28(10):921–942, 2016.
- [251] S. Wang, D. Lo, and L. Jiang. Active code search: Incorporating user feedback to improve code search relevance. In *Proc. ASE*, pages 677–682, 2014.
- [252] S. Wang, D. Lo, and L. Jiang. Autoquery: automatic construction of dependency queries for code search. *ASE*, 23(3):393–425, Sep 2016.
- [253] Y. Wang, L. Wang, Y. Li, D. He, and T. Liu. A theoretical analysis of NDCG type ranking measures. In *Proc. COLT*, pages 25–54, 2013.
- [254] F W Warr and M P Robillard. Suade: Topology-Based Searches for Software Investigation. In *Proc. ICSE*, pages 780–783, 2007.

- [255] M Wen, R Wu, and S C Cheung. Locus: Locating bugs from software changes. In *Proc. ASE*, pages 262–273, 2016.
- [256] M. White, M. Linares-Vásquez, P. Johnson, C. Bernal-Cárdenas, and D. Poshyvanyk. Generating reproducible and replayable bug reports from android application crashes. In *Proc. ICPC*, pages 48–59, 2015.
- [257] L A Wilson. Using Ontology Fragments in Concept Location. In *Proc. ICSM*, pages 1–2, 2010.
- [258] C P Wong, Y Xiong, H Zhang, D Hao, L Zhang, and H Mei. Boosting Bug-Report-Oriented Fault Localization with Segmentation and Stack-Trace Analysis. In *Proc. ICSME*, pages 181–190, 2014.
- [259] E Wong, J Yang, and L Tan. AutoComment: Mining Question and Answer sites for Automatic Comment Generation. In *Proc. ASE*, pages 562–567, 2013.
- [260] R. Wu, H. Zhang, S. Kim, and S. Cheung. Relink: Recovering links between bugs and changes. In *Proc. ESEC/FSE*, pages 15–25, 2011.
- [261] M. Wursch, G. Ghezzi, G. Reif, and H. C. Gall. Supporting developers with natural language queries. In *Proc. ICSE*, pages 165–174, 2010.
- [262] X. Xia, L. Bao, D. Lo, and S. Li. “automated debugging considered harmful” considered harmful: A user study revisiting the usefulness of spectra-based fault localization techniques with professionals using real bugs from large systems. In *Proc. ICSME*, pages 267–278, 2016.
- [263] X. Xia, L. Bao, D. Lo, P. S. Kochhar, A. E. Hassan, and Z. Xing. What do developers search for on the web? *EMSE*, 22(6):3149–3185, 2017.
- [264] T Xie and J Pei. MAPO: Mining API Usages from Open Source Repositories. In *Proc. MSR*, pages 54–57, 2006.
- [265] J Yang and L Tan. Inferring Semantically Related Words from Software Context. In *Proc. MSR*, pages 161–170, 2012.
- [266] J. Yang and L. Tan. Swordnet: Inferring semantically related words from software context. *EMSE*, 19(6):1856–1886, 2014.
- [267] J. Yao, B. Cui, L. Hua, and Y. Huang. Keyword Query Reformulation on Structured Data. In *Proc. ICDE*, pages 953–964, 2012.
- [268] X Ye, H Shen, X Ma, R Bunescu, and C Liu. From Word Embeddings to Document Similarities for Improved Information Retrieval in Software Engineering. In *Proc. ICSE*, pages 404–415, 2016.
- [269] Xin Ye, Razvan Bunescu, and Chang Liu. Learning to rank relevant files for bug reports using domain knowledge. In *Proc. FSE*, pages 689–699, 2014.

- [270] K C Youm, J Ahn, J Kim, and E Lee. Bug Localization Based on Code Change Histories and Bug Reports. In *Proc. APSEC*, pages 190–197, 2015.
- [271] H. Yu, W. Song, and T. Mine. Apibook: An effective approach for finding apis. In *Proc. Internetware*, pages 45–53, 2016.
- [272] T Yuan, D Lo, and J Lawall. Automated Construction of a Software-Specific Word Similarity Database. In *Proc. CSMR-WCRE*, pages 44–53, 2014.
- [273] S. Zamani, S. Peck Lee, R. Shokripour, and J. Anvik. A noun-based approach to feature location using time-aware term-weighting. *IST*, 56(8):991 – 1011, 2014.
- [274] F. Zhang, H. Niu, I. Keivanloo, and Y. Zou. Expanding queries for code search using semantically related api class-names. *TSE*, 44(11):1070–1082, 2018.
- [275] Y. Zhang, W. Zhang, J. Pei, X. Lin, Q. Lin, and A. Li. Consensus-based ranking of multivalued objects: A generalized borda count approach. *TKDE*, 26(1):83–96, 2014.
- [276] J. Zhou, H. Zhang, and D. Lo. Where should the bugs be fixed? more accurate information retrieval-based bug localization based on bug reports. In *Proc. ICSE*, pages 14–24, 2012.
- [277] Y Zhou, Y Tong, R Gu, and H Gall. Combining Text Mining and Data Mining for Bug Report Classification. In *Proc. ICSME*, pages 311–320, 2014.
- [278] T Zimmermann, N Nagappan, and A Zeller. Predicting Bugs from History. In *Software Evolution*, pages 69–88. Springer, 2008.

# APPENDIX A

## REPLICATION PACKAGES

### A.1 STRICT

It accepts a change request as a query, identifies suitable keywords from the request texts using graph-based term weighting algorithms, and then delivers an improved, reformulated query for concept location.

- **Project website:** <http://www.usask.ca/~mor543/strict>
- **GitHub repository:** <https://github.com/masud-technope/STRICT-Replication-Package>

### A.2 ACER

It accepts a given query as input, identifies complementary keywords from the relevant source code documents (retrieved by the query) using a graph-based term weighting method, and then delivers an improved, reformulated search query for concept location.

- **Project website:** <http://www.usask.ca/~mor543/acer>
- **GitHub repository:** <https://github.com/masud-technope/ACER-Replication-Package-ASE2017>

### A.3 BLIZZARD

It accepts a bug report as a query, employs appropriate methodologies or algorithms for keyword selection from the report texts based on the quality of report (e.g., noisy, poor), and then delivers an improved, reformulated search query for bug localization.

- **Project website:** <http://www.usask.ca/~mor543/blizzard>
- **GitHub repository:** <https://github.com/masud-technope/BLIZZARD-Replication-Package-ESECFSE2018>
- **ACM archive:** <https://dl.acm.org/citation.cfm?id=3277001>

### A.4 BLADER

It accepts a poor bug report as a query, identifies complementary keywords from the relevant source code documents based on the clustering tendency between the query and the candidate keywords, and then delivers an improved, reformulated search query for the bug localization.

- **GitHub repository:** <https://github.com/masud-technope/BLADER-ICSE2019-Replication-Package>

### A.5 RACK

It accepts a free-form search query on a programming task, expands the query with relevant API classes that are carefully mined from the crowd generated knowledge of Stack Overflow Q&A site, and then delivers an improved, reformulated query for Internet-scale code search.

- **Project website:** <http://www.usask.ca/~mor543/rack>
- **GitHub repository:** <https://github.com/masud-technope/RACK-Replication-Package>
- **Tool demonstration:** <https://youtu.be/50Fbx8g5eXk>



## A.6 NLP2API

It accepts a free-form search query on a programming task, expands the query with relevant API classes that are carefully collected based on query-API semantic distance analysis, and then delivers an improved, reformulated query for Internet-scale code search.

- **Project website:** <http://www.usask.ca/~mor543/nlp2api>
- **GitHub repository:** <https://github.com/masud-technope/NLP2API-Replication-Package>

## A.7 Other PhD Projects

- Other completed PhD projects & associated materials

# APPENDIX B

## BUGDOCTOR

We present six studies – STRICT (Chapter 3), ACER (Chapter 4), BLIZZARD (Chapter 5), BLADER (Chapter 6), RACK (Chapter 7) and NLP2API (Chapter 8)– that support the software developers in various code searches with automated query reformulations. Although each of these studies has produced individual tool with command line user interfaces, we further combine them, and package them into a single Eclipse IDE plug-in namely **BugDoctor**. The plug-in can be downloaded, installed and easily integrated into the developer’s work environment (e.g., IDE). In this appendix, we present the download link and several use case scenarios (e.g., concept location, bug localization, Internet-scale code search) of our developed tool.

### B.1 Download

The Eclipse plug-in and its dependencies can be downloaded from the BugDoctor website. Please consult with the README file for detailed installation guidelines. An overview on BugDoctor can also be found at YouTube: <https://www.youtube.com/watch?v=RPMBR0Ktxks>.

### B.2 Configuration Setup

Once the plug-in is installed successfully, the default configurations (Fig. B.1) should be modified. The configuration window can be found here: **Window > Preferences > Ant > BugDoctor**

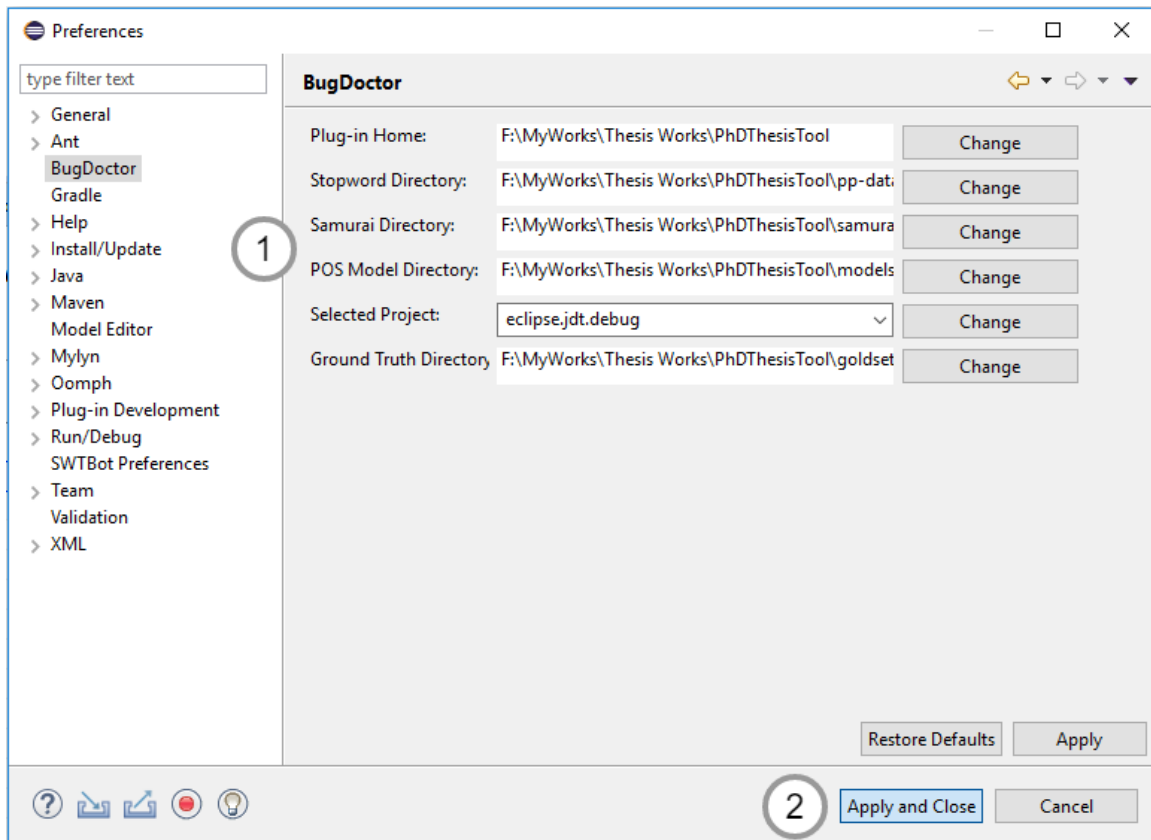


Figure B.1: Setting up custom configurations for BugDoctor

## B.3 Enabling BugDoctor in the IDE

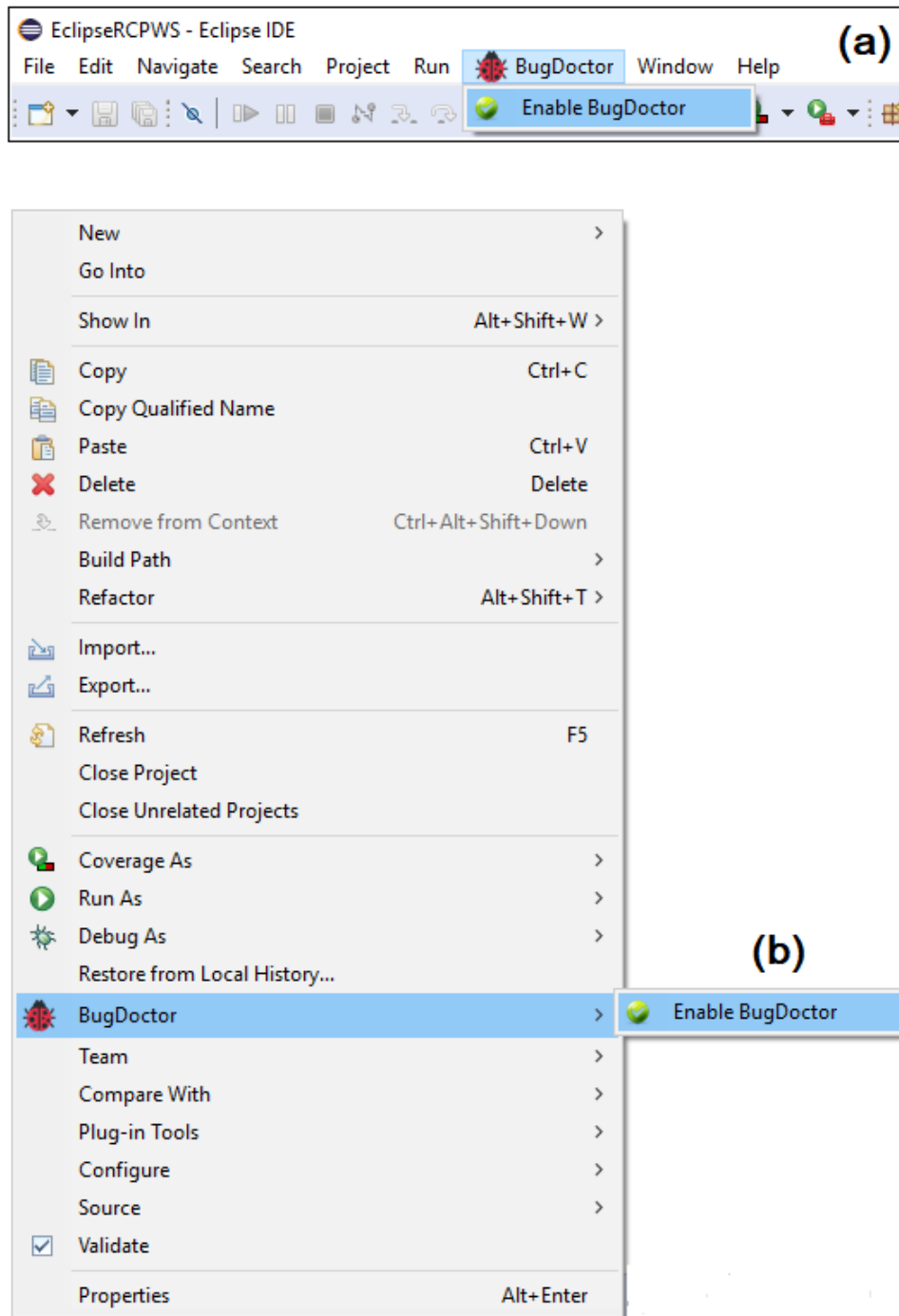
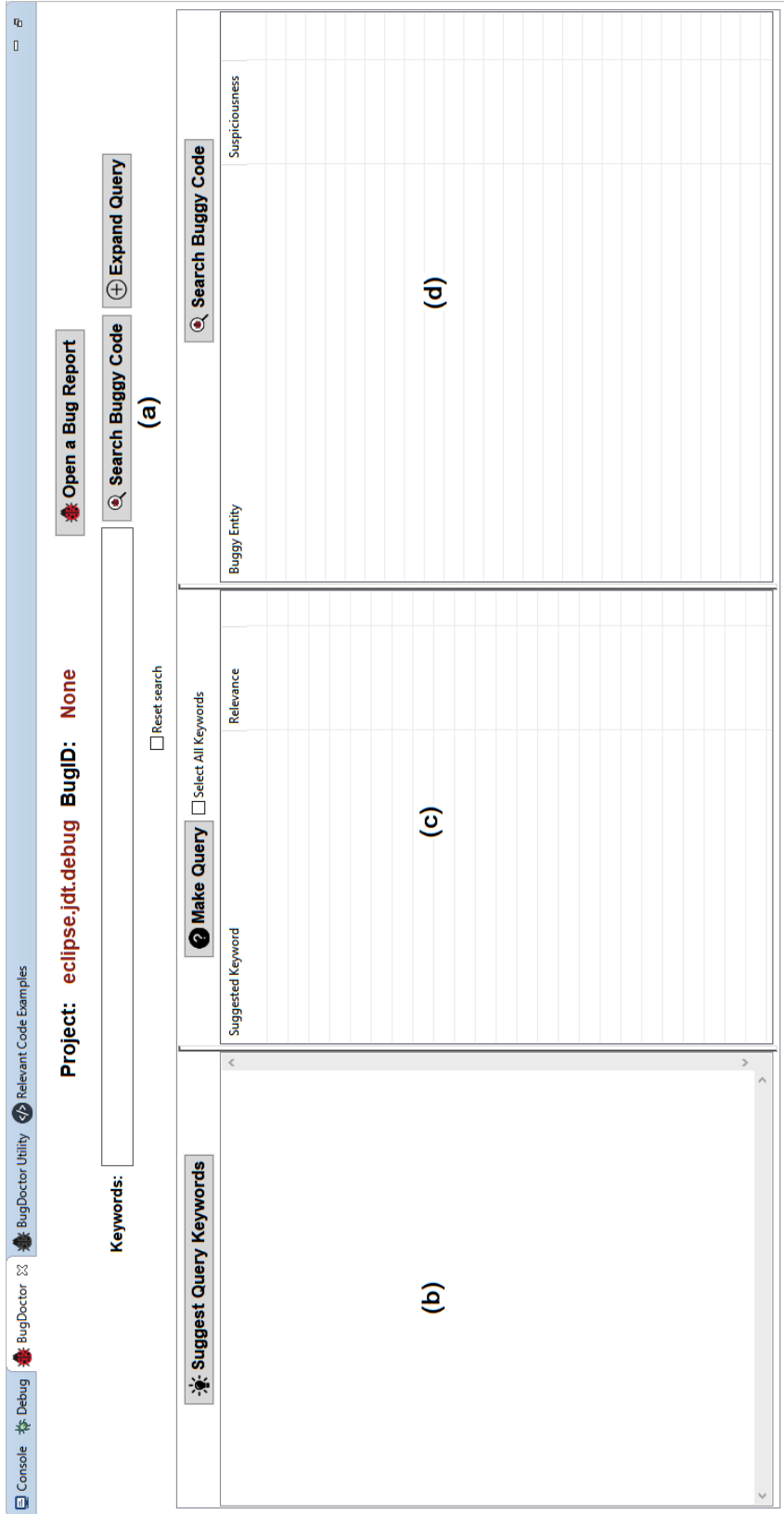


Figure B.2: Enabling BugDoctor with (a) main menu option and (b) context menu option

## B.4 BugDoctor User Interfaces

BugDoctor has three different windows: BugDoctor Dashboard (Fig. B.3), BugDoctor Utility Dashboard (Fig. B.4), and Relevant Code Example Dashboard (Fig. B.5)



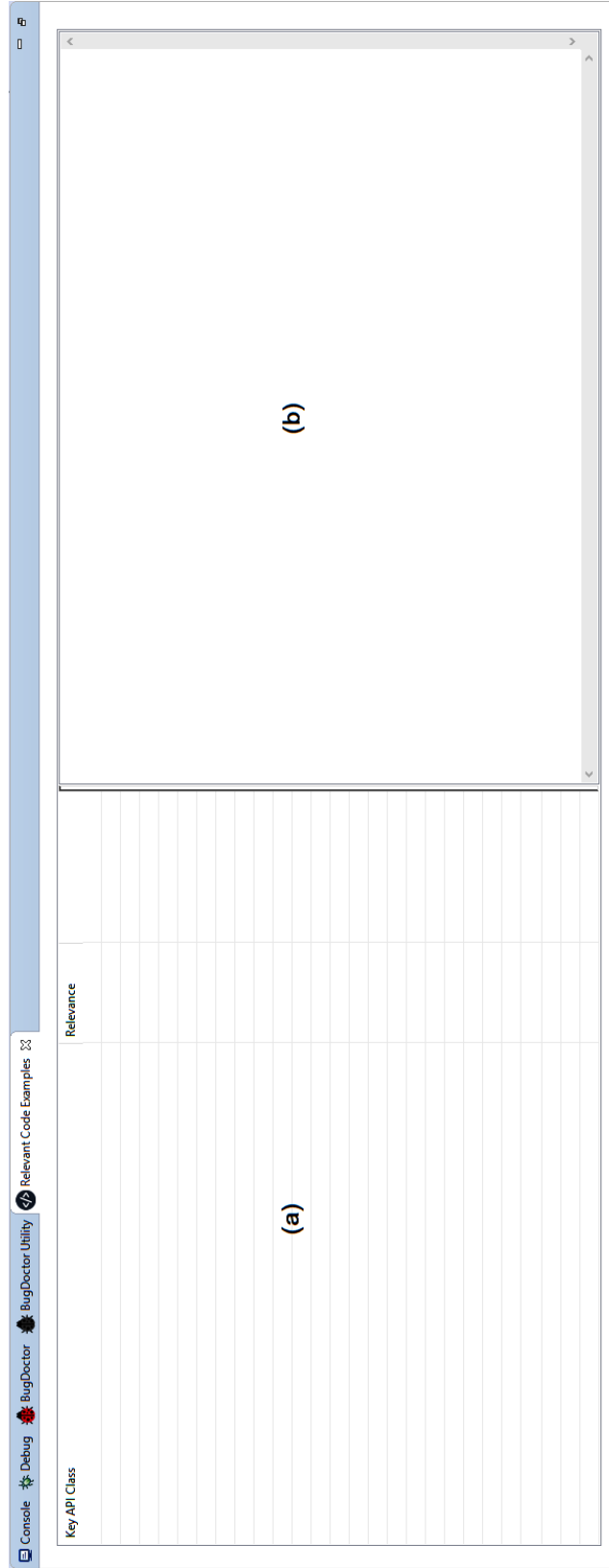
**Figure B.3:** BugDoctor Dashboard: (a) Query execution panel, (b) Bug report panel, (c) Query reformulation panel, and (d) Code search results panel

The screenshot shows the BugDoctor Utility dashboard. At the top, there is a navigation bar with "Tasks", "BugDoctor Utility", and "Relevant Code Examples". Below this is a "Keywords:" section with an input field and a "Reset Search" button. A checkbox labeled "Use GitHub" is checked. To the right, there are three buttons: "Get Relevant APIs", "Search Relevant Code", and "Show Top-K Code Examples".

The main area is divided into two sections. Section (b) is a table titled "Expand Query" with columns for "API Class", "KAC", "KPAC", "KKC", and "Relevance". The table is currently empty. Section (c) is a large text area containing the label "(c)".

At the bottom right, there is a small text label: "[Original source location on the web]".

**Figure B.4:** BugDoctor Utility Dashboard: (a) API suggestion & query execution panel, (b) Query expansion panel, and (c) Code viewer



**Figure B.5:** Code Example Dashboard: (a) Top-K relevant code examples, and (b) Code viewer

## B.5 Loading an Issue Report (e.g., Change Request, Bug Report)

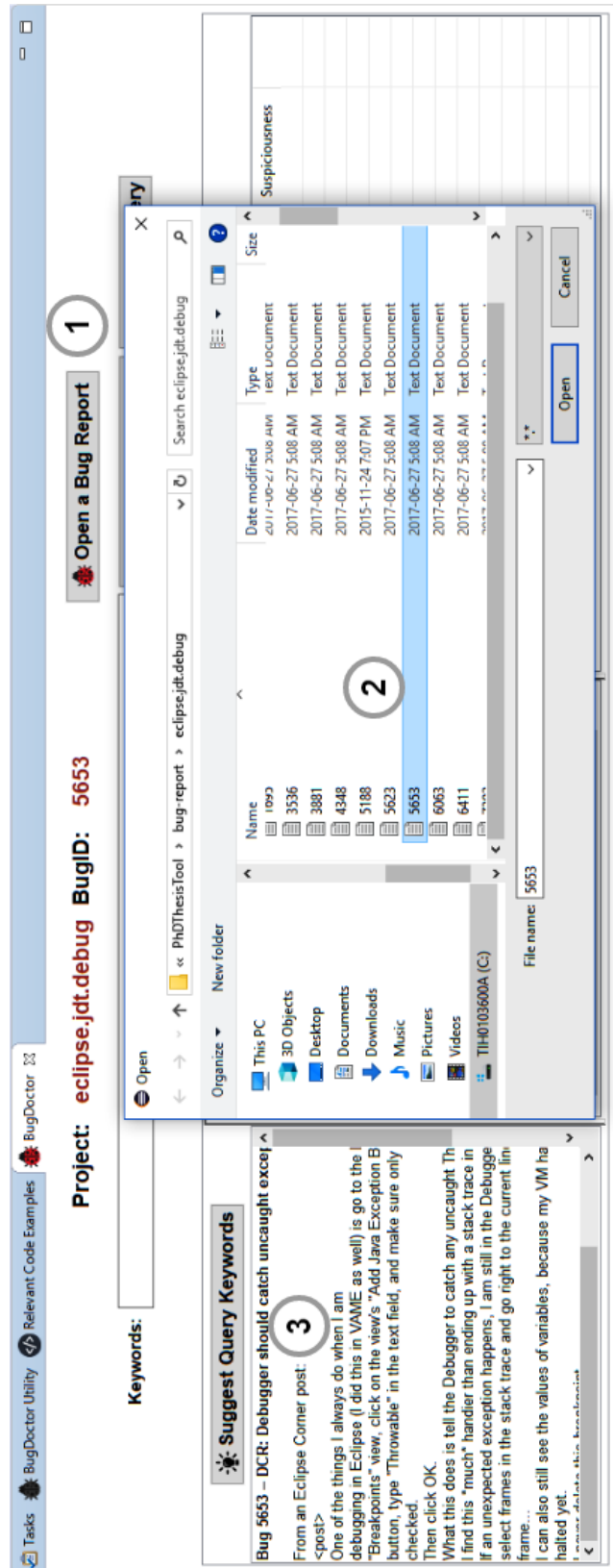


Figure B.6: Loading of an issue report: (1) Click the button, (2) Choose the report, and (3) View the report within the IDE

## B.6 Concept Location with BugDoctor

BugDoctor accepts a change request as a search query, improves it with either query reduction (Fig. B.7) or query expansion (B.9), and then finds out the code that needs to be changed.

**Project:** eclipse.jdt.debug **BugID:** 5653

**Keywords:** Debugger post Uncaught Java exception **6** Eclipse Throwable default catch  Reset search

**1** Open a Bug Report **7** Search **Code** **7** Expand Query

**3** Suggest Query Keywords

**2** Suggested Keyword

- Debugger
- post
- Uncaught
- Java
- exception
- trace
- Eclipse
- Throwable
- default
- catch
- Bug
- DCR
- Debug
- ger
- uncaught

**4** Make Query  Select All Keywords

**5** Relevance

Keyword	Relevance
Debugger	93%
post	87%
Uncaught	81%
Java	75%
exception	68%
trace	62%
Eclipse	56%
Throwable	50%
default	45%
catch	37%
Bug	31%
DCR	25%
Debug	18%
ger	11%
uncaught	8%

**6** Bug 5653 - DCR: Debugger should catch uncaught exception by default

From an Eclipse Corner post:

One of the things I always do when I am debugging in Eclipse (I did this in VAME as well) is go to the Debugger's "Breakpoints" view, click on the view's "Add Java Exception Breakpoint" button, type "Throwable" in the text field, and make sure only "Uncaught" is checked.

Then click OK

What this does is tell the Debugger to catch any uncaught Throwables for me. I find this "catch" handler then ending up with a stack trace in the Console. If an unexpected exception happens, I am still in the Debugger, and I can select frames in the stack trace and go right to the current line in the frame...

I can also still see the values of variables, because my VM has not halted yet.

I never delete this breakpoint.

Every time I run a Java program under the Debugger the "Uncaught Throwable" breakpoint is always there in the background...

protecting me from myself -gim>

The response to this post was as follows:

"That sounds like a great technique, thanks!  
(Why isn't that the default behavior? :)"

**8** Search Buggy Code

Buggy Entity	Suspiciousness
ExceptionEventTest	88%
IJDIPreferencesConstants	86%
AddExceptionTypeDialogExtension	80%
LineNumberTests	78%
JavaExceptionBreakpoint	75%
TopipSpy	75%
JDIDebugPlugin	74%
MacOSXVMInstallType	68%
LaunchingPlugin	66%
AddExceptionAction	64%
JDI TestCase	63%
ExceptionEventImpl	63%
MiscBreakpointTests	63%
JDI DebugOptions	62%
TextResResult	60%
SuspendOnUncaughtExceptionHandler	60%
ThreadReference	58%
LocalVariableTests	55%
ManualSuite	55%
JavaStackTraceConsoleTest	54%

**Figure B.7:** Concept location with query reduction: (1-2) Open a change request, i.e., given query, (3-4) Keyword suggestion, (5-6) Reduced query, (7) Code search, and (8) Located concept within the Top-10 results



Tasks BugDoctor

**Project:** eclipse.jdt.debug **BugID:** 31110

**Keywords:** Debugger Source Lookup does not work with variables

**1** Open a Bug Report

**2** Suggest Query Keywords

**3** Make Query

**4** Search

**5** Search Buggy Code

**Expand Query**

Reset search

Select All Keywords

**Suggest Query Keywords**

**Bug 2** Debugger Source Lookup does not work with variables

In the Debugger Source Lookup dialog I can also select variables for source lookup.  
(Advanced...  
> Add Variables).  
I selected the variable which points to the archive containing the source file for the type, but the debugger still claims that he cannot find the source.  
Eclipse Build 200212181304 (M4).

Suggested Keyword	Relevance	Suspiciousness
Buggy Entity		99%
ProjectClasspathVariableTests		95%
SourceLocationTests		94%
DetailPaneManagerTests		93%
AutomatedSuite		90%
DirectorySourceLookupTests		89%
SourceLookupTests		89%
ArchiveSourceLookupTests		88%
JavaSourceLookupDirector		88%
AbstractJavaLaunchConfigurationDelegate		88%
JavaSourceLookupDialog		88%
JavaSourceLookupParticipant		87%
JavaSourceLookupTab		87%
ProjectSourceContainerTests		87%
ClasspathContainerSourceContainerBrowser		86%
WorkspaceSourceContainerTests		86%
JavaSourcePathComputer		86%
ExternalArchiveSourceContainerTests		86%
FolderSourceContainerTests		86%
DefaultSourceContainerTests		86%
DirectorySourceContainerTests		86%

**Figure B.8:** Concept location with baseline query, (1-3) Selection of report title as a baseline query, (4) Code search, and (5) Concept not located within the Top-10 results

Tasks BugDoctor

Project: **eclipse.jdt.debug BugID: 31110** **Open a Bug Report**

Keywords: **Debugger Source Lookup work variables** **5** type lookup launch test debug problem path **Search** **6** Code **Expand Query** **2**

**1** **Suggest Query Keywords**  
 In the Debugger Source Lookup dialog I can also select variables for source lookup.  
 (Advanced...  
 > Add Variables).  
 I selected the variable which points to the archive containing the source file for the type, but the debugger still claims that he cannot find the source.  
 Eclipse Build 200212181304 (M4).

**4** **Make Query**  Reset search  Select All Keywords

Suggested Keyword

- Debugger
- Source
- Lookup
- work
- variables
- source
- type
- lookup
- launch
- test
- debug
- problem
- path
- def
- java

**3**

Relevance

93%
87%
81%
75%
68%
62%
56%
50%
45%
37%
31%
25%
18%
11%
5%

**7**

**Search Buggy Code**

Buggy Entity	Suspiciousness
SourceLookupTests	96%
AbstractJavaLaunchConfigurationDelegate	94%
SourceLookupBlock	94%
JarSourceLookupTests	90%
JavaDebugUtils	87%
DirectorySourceLookupTests	86%
JavaRuntime	85%
JavaSourceLocator	82%
ExternalArchiveSourceContainerTests	82%
JavaSourceLookupDialog	82%
JavaSourcePathComputer	81%
LauncherMessages	81%
FolderSourceContainerTests	81%
JavaSourceLookupTab	80%
AutomatedSuite	80%
ClasspathContainerSourceContainer	78%
ClasspathContainerSourceContainerBrowser	77%
IJavaLaunchConfigurationConstants	77%
ArchiveSourceLocation	77%
JavaSourceLookupUtil	76%

**Figure B.9:** Concept location with query expansion : (1) Selection of report title as a given query, (2-3) Query expansion, (4-5) Expanded query, (6) Code search, and (7) Concept located within the Top-10 results

## B.7 Bug Localization with BugDoctor

BugDoctor accepts a bug report as a search query, improves it with either query reduction (Fig. B.10) or query expansion, and then finds out the buggy code that needs to be fixed.

**Project:** eclipse.jdt.debug BugID: 31637

**Keywords:** Bug - should be able to cast null NullPointer on toString JDValue EvaluationThread E

**1** Open a Bug Report

**2** Suggest Query Keywords

**3** Make Query

**4** Bug - should be able to cast null NullPointerException  
toString  
JDValue  
EvaluationThread  
Runnable  
String  
access  
evaluation  
run  
execute  
Thread  
runEvaluation

**5** Search Buggy Code

**6** Search Buggy Code

**7** Analysis for bug fixing

**8** Keyword suggestion

**9** Localized buggy code

```

public cast(int typeId, String baseType, int dimension, int start) {
    super(start);
    typeId = baseTypeId;
    baseType = baseType;
    dimension = dimension;
}

@Override
public void execute() throws CoreException {
    JDValue value = popValue();
    if (value instanceof JDValuePrimitiveValue) {
        JDValuePrimitiveValue primitiveValue = (JDValuePrimitiveValue) value;
        cast(primitiveValue.getTypeId());
        case T_Double:
            push(newValue(primitiveValue.getDoubleValue()));
            break;
        case T_Float:
            push(newValue(primitiveValue.getFloatValue()));
            break;
        case T_Long:
            push(newValue(primitiveValue.getLongValue()));
            break;
        case T_Int:
            push(newValue(primitiveValue.getIntValue()));
            break;
        case T_Short:
            push(newValue(primitiveValue.getShortValue()));
    }
}
    
```

**Figure B.10:** Bug localization with query reduction: (1-2) Open a bug report, i.e., given query, (3-4) Keyword suggestion, (5-6) Reduced query, (7) Code search, and (8) Localized buggy class as the topmost result, and (9) Analysis for bug fixing

## B.8 Code Example Search with BugDoctor

BugDoctor accepts a programming task description as a search query, improves it using relevant API classes, and then delivers the relevant code examples that implement the given task.

The screenshot shows the BugDoctor web interface with the following components and annotations:

- 1**: Search query: "How to parse HTML in Java?"
- 2**: "Get Relevant APIs" button
- 3**: List of API classes with relevance percentages:
 

API Class	KAC	KPAC	KKC	Relevance
<input checked="" type="checkbox"/> Document	100%	100%	42%	51%
<input checked="" type="checkbox"/> Jsoup	47%	66%	42%	46%
<input checked="" type="checkbox"/> Element	63%	59%	100%	40%
<input type="checkbox"/> IOException	78%	51%	100%	34%
<input type="checkbox"/> File	89%	1%	100%	28%
<input type="checkbox"/> ArrayList	26%	48%	100%	25%
<input checked="" type="checkbox"/> Pattern	36%	48%	31%	23%
<input checked="" type="checkbox"/> Elements	57%	1%	40%	17%
<input type="checkbox"/> Map				11%
<input type="checkbox"/> Matcher				1%
- 4**: "Expand Query" button
- 5**: "Use GitHub" checkbox
- 6**: "Search Relevant Code" button
- 7**: Retrieved code example:
 

```
private void prepareReport(String pattern) throws IOException {
    Document jenkinsResults = Jsoup.connect("https://builds.apache.org/j
    Elements unitTestFailureReport = jenkinsResults.select("a.model-link
    for (Element link : unitTestFailureReport) {
        if (link.text().contains(pattern)) {
            String[] split = link.text().split(" ", module );
            String[] split1 = split[1].split(" has ");
            String moduleName = split1[0];
            links.put(moduleName, link);
        }
    }
    scanWikiAndGenerate();
}
```
- 8**: "Original source" link
- 9**: "Show Top-K Code Examples" button

**Figure B.11:** Code example search with query expansion: (1) Given programming task, i.e., given query, (2-3) Relevant API suggestion, (4-5) Expanded query, (6) Code example search, (7) Retrieved code example, (8) Original code location on the web, and (9) Click the button for Top-K code examples

Tasks BugDoctor Utility Relevant Code Examples

Key/API Class	Relevance
Jsoup Document Elements Element	100%
Document Elements Element Jsoup	99%
Pattern	96%
<b>Document Elements Element Jsoup</b>	<b>96%</b>
Document	93%
Jsoup Document Elements Element Pattern	93%
Elements Element	85%
Document Jsoup Elements Element	85%
Elements	79%
Elements Element	79%
Elements Element	78%
Document	78%

**1**

```

public static List<String> parseFileList(String baseUrl, InputStream str
try {
    URI baseUrl = new URI(baseUrl);
    // to make debugging easier, start with a string. This is assumi
    // assumption.
    String content = IOUtils.toString(stream, "utf-8");
    Document doc = Jsoup.parse(content, baseUrl);
    Elements links = doc.select("a[href]");
    Set<String> results = new HashSet<String>();
    for (Element link : links) {
        /*
        * The abs:href loses directories, so we deal with absol
        */
        String target = link.attr("href");
        if (target != null) {
            String clean = cleanLink(baseUrl, target);
            if (isAcceptableLink(clean)) {
                results.add(clean);
            }
        }
    }
    return new ArrayList<String>(results);
} catch (URISyntaxException e) {
    throw new TransferFailedException("Unable to parse as base URI:
catch (IOException e) {

```

**2**

Figure B.12: Relevant code examples: (1) Top-K code examples, and (2) Code example viewer