SWordNet: Inferring Semantically Related Words from Software Context

by

Jinqiu Yang

A thesis
presented to the University of Waterloo
in fulfillment of the
thesis requirement for the degree of
Master of Applied Science
in
Electrical and Computer Engineering

Waterloo, Ontario, Canada, 2013

I hereby declare that I am the sole author of this thesis. This is a true copy of the thesis, including any required final revisions, as accepted by my examiners.

I understand that my thesis may be made electronically available to the public.

Abstract

Code search is an integral part of software development and program comprehension. The difficulty of code search lies in the inability to guess the exact words used in the code. Therefore, it is crucial for keyword-based code search to expand queries with semantically related words, e.g., synonyms and abbreviations, to increase the search effectiveness. However, it is limited to rely on resources such as English dictionaries and WordNet to obtain semantically related words in software, because many words that are semantically related in software are not semantically related in English. On the other hand, many words that are semantically related in English are not semantically related in software.

This thesis proposes a simple and general technique to automatically infer semantically related words (referred to as rPairs) in software by leveraging the context of words in comments and code. In addition, we propose a ranking algorithm on the rPair results and study cross-project rPairs on two sets of software with similar functionality, i.e., media browsers and operating systems. We achieve a reasonable accuracy in nine large and popular code bases written in C and Java. Our further evaluation against the state of art shows that our technique can achieve a higher precision and recall. In addition, the proposed ranking algorithm improves the rPair extraction accuracy by bringing correct rPairs to the top of the list. Our cross-project study successfully discovers overlapping rPairs among projects of similar functionality and finds that cross-project rPairs are more likely to be correct than project-specific rPairs. Since the cross-project rPairs are highly likely to be general for software of the same type, the discovered overlapping rPairs can benefit other projects of the same type that have not been anaylyzed.

Acknowledgements

I would like to take this opportunity to express my biggest thanks to my supervisor Prof. Lin Tan. From her, I get numerous supports and encouragement in every aspect. I would like to thank for her insightful guidance and unwavering supports. She is always available when I ask for help or advice. I learned a lot from Lin, including conducting research, time management, presentations skills and how to communicate and collaborate with other people. The things I learned from Lin will extremely benefit my future development.

I am thankful to readers of the thesis, Prof. Ladan Tahvildari and Prof. Mahesh V. Tripunitara, for spending their valuable time in reviewing my thesis and providing valuable comments.

I would like to thank all of our research group members. Many thanks for the happy time and discussions in our reading group seminars.

Lastly, I would like to acknowledge my parents and my best friend, partner, and the most patient audient, Tsehsun Chen. They have been always supporting me under any circumstances. From them, I perceive the power of love and concern, which have been and will be with me.

Table of Contents

List of Tables **vii**

List of Figures **ix**

1 Introduction **1**

2 Related Work **6**

2.1 Extracting Semantically Related Word Pairs 6

2.2 Code Search 8

2.3 Analysis of Natural-Language Text for Software 8

3 Basic rPair Extraction **9**

3.1 Parsing Comments and Code 11

3.2 Clustering Comments and Code 11

3.3 Extracting Semantically Related Word Pairs 12

3.4 Refining Semantically Related Word Pairs 13

4 Improved Version of rPair Extraction **14**

4.1 An Improved Similarity Measure 14

4.1.1 The *idf* Weight 14

4.1.2 The New Similarity Measure Definition 15

4.2 The Ranking Function 15

4.3 Studying Cross-Project rPairs 16

5 Experimental Methods **18**

5.1 Experiment: rPair Extraction Accuracy and Comparison with WordNet and a Dictionary 18

5.2 Experiment: Search-Related Evaluation 19

5.3 Experiment: Sensitivity Evaluation 21

5.4 Experiment: Ranking Evaluation 21

5.5 Experiment: Cross-Project rPairs Study 22

5.6 Threats to Validity and Limitations 22

6 Evaluation Results **24**

6.1 rPair Extraction Results 24

6.2 Search-Related Results 27

6.3 Sensitivity Results 31

6.4 Ranking Results 32

6.5 Cross-Project rPair Results 33

7 Conclusions **35**

References **36**

List of Tables

3.1 Learning semantically related word pairs from context. The comment and code examples are real comments and code segments from the nine code bases used in our evaluation. 10

5.1 Evaluated Software. LOComment is lines of comments. *The versions of iReport, jBidWatcher, javaHMO, jajuk are the same as [50]. The dates when NetBSD and OpenBSD were checked out from the version control systems are shown instead of the version numbers. 19

6.1 rPair Extraction Results. Dic denotes Merriam-Webster English Dictionary and Thesaurus [35] and the computer specific dictionary [10]. The margin of error is calculated with 95% confidence level, which is to express the random sampling error in the extraction accuracy. The average accuracy of comment-comment is 56.9%, 33.3% for code-code and 16.5% for comment-code. 25

6.2 rPairs Extraction Results. This table shows the percentage of correct rPairs distributed among five categories for nine projects. Majority of the correct rPairs (42.5-100%) belong to the 'Related' category. 26

6.3 Search-Related results based on function gold set. We search by "*verb*noun*", "*noun*verb*" and combinations of alternative words for 'verb' and 'noun'. CTX is our context-based technique, CTX_T is CTX with transitive rPairs, and SWUM denotes the previous work by [16]. The function gold set is from [49]. . 28

6.4 Search-Related results on rPairs gold set. CTX is our context-base technique, CTX_T is CTX with transitive rPairs, and SWUM denotes the previous work by [16]. 29

6.5 The accuracy of rPair extraction with ranking. Top 10 can achieve the highest accuracy of 60-100%, top 30 achieve the accuracy of 66.7-83.3% and top50 has the accuracy of 60-80%. 32

6.6 Cross-project rPair Results. 33

List of Figures

1.1 Overview of Our Approach 5

Chapter 1

Introduction

Code search is an integral part of software development; developers spend up to 19% of their development time on code search [27]. It becomes more difficult for one developer to understand and remember every piece of a software project, as software becomes larger and more complex, software is typically developed by hundreds of or thousands of programmers across decades, and developers frequently join and depart from the software development process. In order to find relevant code segments, code search is becoming a crucial part of software development and program comprehension.

The search for relevant code segments is difficult, because there is a small chance (10-15%) that developers guess the exact words used in the code [13]. For example, if developers want to find methods that disable interrupts in the Linux kernel, a simple regular expression based search "*disable*interrupt*"[1] will miss the functions "`disable_irq`" and "`mask_irq`". Both functions disable interrupts. The problem is the mismatches between the words *interrupt* and *irq* and between the words *disable* and *mask*. Similarly, if we want to find functions which add auctions in jBidWatcher and search for "*add*auction*" in the code, the method "`AuctionsManager.newAuctionEntry(String)`" will not be returned, although it is related to adding an auction entry.

Researchers proposed to expand search queries with semantically related words (e.g., synonyms and abbreviations) for more effective searches [50]. However, leveraging an English dictionary [35] and WordNet [44] for obtaining semantically related words is limited in the software domain for two reasons. First, many words that are semantically related in *software* are not

[1] It is possible to perform a relaxed search to find method names that contain either the word *disable* or the word *interrupt*, but such an approach generally retrieves too many irrelevant matches to be useful.

semantically related in *English*. In the previous example, the words *disable* and *mask* are not related words either in an English dictionary [35] or WordNet [44]. Similarly, *interrupt* and *irq* are not semantically related in the English dictionary or WordNet. A recent study evaluated six well known techniques for discovering semantically related words in English and showed that these techniques are limited in identifying semantically related words in software [53]. The best technique needs to find over 3,000 pairs of words in order to discover 30 out of the 60 semantically related word pairs in the gold set.

Second, many words that are semantically related in *English* are not semantically related in *software*. For example, the words *disable* and *torture* are semantically related in English, but not semantically related in the interrupt context.

If we can automatically discover semantically related words from software, it would not only improve search tasks, but also benefit other software engineering tasks. For example, aComment [56] leverages semantically related words to find comments that have similar meanings in order to check these comments against source code to detect bugs. Currently, aComment requires its users to manually specify synonyms and paraphrases, which is challenging since it requires the users to have domain knowledge about the target software. In addition, the ad hoc process is likely to miss important synonyms and paraphrases. An automated approach can potentially discover more synonyms and paraphrases and reduce the manual effort required.

Therefore, we propose to automatically identify semantically related words by leveraging the *context* of words in comments and code. This includes relations such as synonyms, antonyms, abbreviations, related words, etc., all of which are useful for code search. We use *semantically related word pairs* or the shorter *rPairs* to denote a pair of semantically related words and phrases. Our intuition is that if two words or phrases are used in the same context in comment sentences or identifier names, then they likely have syntactic and semantic relevance. For example, by examining the two comment sentences from the Linux kernel—"`Disable all interrupt sources.`" and "`Disable all irq sources.`", we can learn that the words *interrupt* and *irq* are likely to be related because both words appear in the same context. In this particular case, the two words have the same meaning. Similarly, from the functions, for example, "`void mask_all_interrupts()`" and "`void disable_all_interrupts(...)`", we can infer that the word *mask* and the word *disable* form an rPair in this context. In addition to learning nouns and verbs that have similar meanings, we can learn adjectives with similar meanings. For example, we can infer that the two adjectives *disabled* and *off* have the same meaning from the following two comments—"`Must be called with interrupts disabled.`" and "`It MUST be called with interrupts off.`".

Shepherd et al. [50, 51] extract verb-DO (Direct Object) pairs from software which can be

leveraged to identify semantically related words. For example, if they discover verb-DO pairs *(add | element)* and *(find | element)* in iReport, they would suggest the word *find* to users to expand their query "add element" in iReport [50], because (*add*, *find*) are considered semantically related. This work differs from the previous work mainly in the following aspects. First, the previous work relies on heuristics regarding the naming convention and the structure of code identifiers and comments. For example, they use different heuristics to extract the DO from a method name, depending on whether a verb exists in the method name, where the verb is, and what the verb is. Such heuristics are manually designed by the authors and may not generalize if the naming convention or structure is not followed. Our technique *requires no heuristics about the naming convention or the structure of the code identifiers and comments*[2], and can potentially be applied to a broader spectrum of code bases.

Second, the previous technique leverages Natural Language Processing (NLP) techniques, such as part-of-speech (POS) tagging and chunking, which are trained from general English text such as the Wall Street Journal, not from software. When applied to the software domain, these techniques can cause inaccuracies in rPair extraction. For example, it would fail to identify the verb-DO pair from "`newParameter()`", as *new* is a noun in English. But in the software context, *new* is commonly used as a verb to refer to creating memory for a new object. This inaccuracy prevents the previous techniques from discovering the rPair *(new, add)* that our technique can discover, because our technique ignores the part of speech. The detailed comparison is discussed in Chapter 2.

This thesis makes the following contributions.

- We propose a context-based approach to automatically infer semantically related words by leveraging the *context* of words in comments and code. Our technique can be used as a building block for many other software engineering tasks including code search [18, 50] and software bug detection [56].
- The proposed technique identifies semantically related words with a reasonable accuracy in nine large and popular code bases written in C and Java—the Linux kernel, Apache HTTPD Server, Apache Commons Collections, iReport, jBidWatcher, javaHMO, jajuk, NetBSD and OpenBSD. We classify the semantically related word pairs into five categories—*synonym*, *related*, *antonym*, *near antonym*, and *identifier*. The majority of the identified semantically related word pairs cannot be found in WordNet [44], an English dictionary [35] or a computer specific dictionary [10]. The total number of rPairs discovered by our context-based approach ,which ranges from 111 to 108,571 (comment-comment), from 685 to 606,432 (code-code) and up to 10,633 (comment-code), shows the feasibility of our technique.

[2]Except that we break method names into words based on camel case and underscore, which is also used by the previous work

- Our further evaluation against the state of art [16, 50] shows that our overall recall and precision in discovering semantically related word pairs and locating relevant functions is higher. Since automatically expanding queries with inappropriate synonyms may produce worse results than not expanding [53], it may be beneficial to leverage techniques similar to previous work [18, 50] to allow developers to pick from a list of semantically related words. Since our technique has *higher recall* (finds more rPairs or more functions) with *higher precision* (more of the pairs discovered are truly rPairs or more of the functions found are truly relevent to the search task), it can help developers find more relevant code segments and comments, as well as find them more quickly because developers will examine fewer incorrect rPairs.
- We propose and evaluate an algorithm to rank the rPairs from our basic extraction results. Although our basic technique presents new opportunities to discover more semantically related words and improves the accuracy of discovering them, the absolute accuracy is relatively low due to the inherent difficulty of the task. Therefore, we introduce a ranking algorithm to further improve the extraction accuracy. We evaluate our ranking algorithm on the nine projects using three ranking slots, i.e., top 10, top 30 and top 50, and find that the ranking algorithm can significantly improve the accuracy, especially for projects of a large size (i.e., Linux, OpenBSD and NetBSD). We discuss other techniques that can potentially further improve the accuracy in Chapter 6.1.
- We study the cross-project rPairs from two sets of projects. The motivation is that if one rPair occurs in multiple projects, especially projects of similar functionality, the rPair is more likely to be correct and general, and benefit other projects of the same type. Our cross-project study shows that we can find overlapping rPairs among different projects of similar functionality, and that cross-project rPairs are more likely to be correct than project-specific rPairs. Therefore, whether an rPair is a cross-project rPair may be used to improve the ranking algorithm and results. Besides, the cross-project rPairs can supplement the rPairs of the projects with the same kind, which further improves the feasibility of our technique.

Approach Overview

We present the overview of this context-based approach in Figure 1.1. This context-based approach takes code bases, configuration (parameter settings), the reverse mapping dictionary (obtained from an on-line dictionary) and the ranking function as input, and produces two types of output: rPairs (also called "refined rPairs" in Figure 1.1) and the ranked rPairs. The basic rPairs approach has four steps: parsing, clustering, extracting and refining, which is described in Chapter 3. Then the ranking algorithm can further improve the accuracy of our approach, and the details are introduced in Chapter 4.

Thesis Outline The rest of the thesis is organized as follows. Chapter 3 describes our basic approach to learn semantically related word pairs from software context (comments and code).

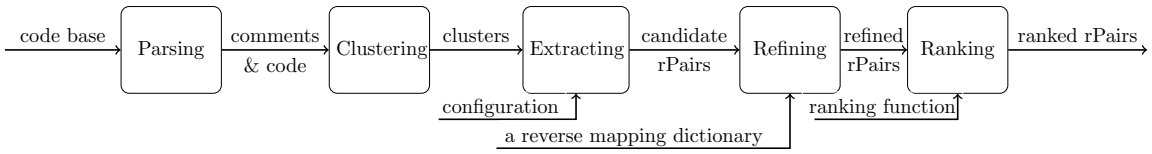

Figure 1.1: Overview of Our Approach

Chapter 4 proposes an algorithm to rank the rPairs learned by our basic approach. Chapter 4.3 presents the motivations of our cross-project rPairs study. Chapter 5 describes how we conduct the experiments. Detailed results and the analysis of the results are provided in Chapter 6. In Chapter 2, a discussion of the related work is presented. Finally, we conclude our findings and discuss about future work in Chapter 7.

Chapter 2

Related Work

2.1 Extracting Semantically Related Word Pairs

The closely related work [50, 51] infers semantically related words in software, and leverages them to build a search tool that outperforms two existing approaches [42]. We have already discussed the main differences between the previous work and our context-based approach in Chapter 1, so we only summarize them here and provide more examples. The previous work relies on manually created heuristics, which may not generalize to other types of software and software written in different programming languages (e.g., non-object-oriented software such as the Linux kernel and HTTPD). Our technique requires no such heuristics and is effective in extracting rPairs from both object-oriented software and non-object-oriented software.

Second, the previous techniques leverage NLP techniques, whose models are trained from general English text, not from software. When applied to the software domain, these models can make mistakes. What is worse, comments and code identifiers are generally incomplete and grammatically incorrect, which may worsen the analysis inaccuracy problem. For example, two sentences with the same structure are analyzed differently because *interrupts* is a English word, while *irqs* is not. OpenNLP [59] (the same tool used by the previous work [50, 51]) tags the comment sentence from the Linux kernel "`called with interrupts disabled`" as

$$called_{<Verb>}\ with_{<Determiner>}\ interrupts_{<Noun>}\ disabled_{<Verb>},$$

and then chunks it as

$$called_{<VerbPhrase>}\ [with\ [interrupts]_{<NounPhrase>}\ ]_{<PrepositionalPhrase>}\ disabled_{<Unknown>}.$$

Since the chunker cannot tag the last word *disabled* with the proper Chunk/Phrase-level tag, the words *called* and *interrupt* will be considered as a verb-DO incorrectly if *called* is considered active voice, or no verb-DO will be found if *called* is considered passive voice (from the previous paper [51], it is unclear whether the heuristics treat *called* as active or passive voice; therefore, we discuss both for completeness). However, a slightly different comment from the Linux kernel "`called with irqs disabled`" will be tagged as

$$called_{<Verb>}\ with_{<Determiner>}\ irqs_{<Adjective>}\ disabled_{<Adjective>},$$

and be chunked as

$$called_{<VerbPhrase>}\ [with\ [irqs\ disabled]_{<NounPhrase>}\ ]_{<PrepositionalPhrase>}.$$

The verb-DO will be (*called* | *irqs disabled*), or no verb-DO is inferred. From these verb-DOs, previous work may consider *interrupts* and *irqs disabled* semantically related words by mistake, or it infers no semantically related words. Our technique can correctly identify *interrupts* and *irqs* as an rPair. This example shows that our technique is robust despite incomplete and grammatically incorrect comments. In addition, NLP analysis adds time complexity, especially POS tagging and chunking.

Since our technique ignores the part of speech, our technique has higher recall than the previous techniques. In addition, since we use similarity measures and consider the full context instead of just the verbs and DOs, our technique can be more precise. For example, from two comments "`initialize the product price`" and "`initialize the user name`", we would not consider the phrases *product price* and *user name* rPairs, since the similarity is only 50%, while the previous technique obtains two verb-DOs that share the verb *initialize*, and considers the two phrases semantically related.

The authors improved the original verb-DO technique by leveraging phrasal concept and more advanced heuristics [16, 19]. The latest implementation uses specialized techniques to address the OpenNLP-related issues. However, the improved technique does not analyze comments, missing the opportunity to detect more rPairs. On the other hand, their approach analyzes return types and parameters, which may find more rPairs. In addition, the restrictions regarding verb-verb match may help filter out false positives. However, our evaluation in Chapter 6.2 shows that our technique has better overall precision and recall in discovering rPairs than their latest implementation [16].

Techniques and resources that discover semantically related words in English [7, 26, 32, 33, 35, 44, 45, 65] are limited in discovering semantically related words in *software* [53]. Other work splits multi-word identifiers and discovers abbreviations in software [17, 29, 30]. Some

other work focuses on discovering the meaning of word phrases of method names [23, 24]. Our technique finds general semantically related word pairs including abbreviations, which is complementary to the previous work. In addition, while the previous work uses statistical analysis, heuristics, and English dictionaries, we leverage the context of words in comments and identifiers.

2.2 Code Search

Keyword-based code search techniques have been developed [16, 18, 19, 41, 42, 50]. Since our technique infers semantically related words from software, it can be leveraged by these search tools to expand queries to further improve the search effectiveness. In addition, since our technique provides not only semantically related words but also the context, our technique could be leveraged by contextual-based search techniques [18] to improve the code search accuracy. Alternatives to keyword-based search include structural search [22, 25, 48, 66]. A recent study [20] investigates how to effectively combine global and local code search techniques.

2.3 Analysis of Natural-Language Text for Software

Previous work analyzes natural-language artifacts such as bug reports [6, 14, 31, 34, 38, 39, 46, 54, 64], comments [55, 56], API documentation [37, 67], identifier names [5, 8] and mailing lists [38] for purposes such as detecting duplicate bug reports, identifying the appropriate developers to fix bugs, improving structure-field names, mining source code descriptions, etc. Recently, by leveraging the fact that programming language is likely to be repetitive and predictable, researchers [21] work on applying statistical language models to code to help software tasks, including code completion, concern location and software mining, etc. This thesis analyzes comments and code to discover semantically related word pairs. Different from some of these studies [5, 8, 21, 55] that use NLP techniques such as POS tagging, chunking, semantic role labelling, and n-gram models, this work chooses not to use these advanced NLP techniques for simplicity, efficiency, and generality. On the other hand, it is conceivable to use NLP techniques to generate NLP-related context to infer semantically related words, which remains our future work.

Chapter 3

Basic rPair Extraction

Our goal is to automatically learn semantically related words and phrases by leveraging the context of words and phrases in comments and code. Examples in Table 3.1 help illustrate how semantically related words and phrases can be learned from comments and code. Column 'Context Type' shows whether the context is from comments or source code: comment-comment indicates that both contexts are from comments; code-code means that both contexts are from source code; and comment-code denotes that one context is from comments, and the other context is from source code. For example, both of the two jajuk comments "`None mounted file for this track.`" and "`None accessible file for this track.`" state that a file associated with the track is missing. Since the words *mounted* and *accessible* are surrounded by the same context, "`None ... file for this track.`", we consider the word pair (*mounted*, *accessible*) an rPair.

Our analysis technique takes a code base and a stopword list as input, and outputs semantically related word pairs. The analysis process consists of four steps: (1) *parsing comments and code*: given a code base, we first parse it to extract all the comment sentences and method names, and convert each of them into a sequence of words; (2) *clustering comments and code*: we cluster the word sequences based on whether they contain at least one common word to reduce the overhead of pairwise comparison in the next step, which is a critical step for our technique to scale up to large code bases such as the Linux kernel; (3) *extracting semantically related word pairs*: we calculate the similarity between a pair of word sequences and extract the corresponding rPairs if the context is similar; and (4) *refining semantically related word pairs*: we finally refine the rPairs by using stemming to remove pairs with the same roots, merging duplicate word pairs, normalizing words, and generating transitive rPairs.

Table 3.1: Learning semantically related word pairs from context. The comment and code examples are real comments and code segments from the nine code bases used in our evaluation.

Context	**Semantically Related Word Pairs**	**Context Type**
`Must be called with interrupts` **disabled**`.` `It MUST be called with interrupts` **off**`.`	(disabled, off)	Comment-Comment
`Disable all` **interrupt** `sources.` `Disable all` **irq** `sources.`	(interrupt, irq)	Comment-Comment
`Always` **called** `with interrupts disabled.` `Always` **invoked** `with interrupts disabled.`	(call, invoke)	Comment-Comment
`None` **mounted** `file for this track.` `None` **accessible** `file for this track.`	(mounted, accessible)	Comment-Comment
Serializes `this map` **to** `the given stream` **Deserializes** `this map` **from** `the given stream`	(serialize, deserialize) (to, from)	Comment-Comment
`Min of spare` **threads** `Min of spare` **daemons**	(thread,daemon)	Comment-Comment
`Empty map with the specified maximum` **size** `Empty map with the specified maximum` **capacity**	(size,capacity)	Comment-Comment
`Gets the value` **associated** `with the key` `Gets the value` **mapped** `with the key specified`	(associate, map)	Comment-Comment
`get a node's` **parent** `get a node's` **left child**	(parent, left child)	Comment-Comment
`An iovec to store the` **headers** `sent` **before** `the file` `An iovec to store the` **trailers** `sent` **after** `the file`	(header, trailer) (before, after)	Comment-Comment
`it was finally rewritten to a` **remote URL** `it was finally rewritten to a` **local path**	(remote, local) (URL, path)	Comment-Comment
mask`_all_interrupts()` **disable**`_all_interrupts(...)`	(mask, disable)	Code-Code
add`Parameter(...)` **new**`Parameter()`	(add, new)	Code-Code
File`TypeFileFilter()` **Directory**`TypeFileFilter()`	(file, directory)	Code-Code
Initialize `signal names` **setup**`_signal_names(...)`	(initialize, setup)	Comment-Code
Alloc `a net device` **add**`_net_device(...)`	(alloc, add)	Comment-Code

3.1 Parsing Comments and Code

We extract all comment blocks from source code files and use a sentence segmentator to split them into comment sentences. Each comment sentence is broken down into a sequence of words by using space as the delimiter. For example, the comment sentence "`Called with interrupts disabled`" is represented as a sequence consisting of four words (case insensitive): <`called`, `with`, `interrupts`, `disabled`>. Similarly, we extract method names from source code files, and split them into words based on camel case or underscore. To minimize the dependency on naming convention and code structure related heuristics, our analysis ignores return types and parameters.

A sentence segmentator for English sentences does not work well for code comments mainly because incorrect punctuation is common in comments. Therefore, in addition to regular sentence delimiters, i.e., "!", "?", and ";", we use "." and spaces together as sentence delimiters instead of using "." alone, and consider an empty line and the end of a comment as the end of a sentence [56].

In order to discover semantically related identifiers and avoid duplicate analysis, we do not break identifiers in comments into multiple words based on camel case or underscore. For example, we can learn that the `apr_pool_clear` and `apr_pool_destroy` are semantically related methods in HTTPD from comments "`If you do not have apr_pool_clear in a wrapper`" and "`If you do not have apr_pool_destroy in a wrapper`".

3.2 Clustering Comments and Code

It is expensive to conduct pairwise comparison for a large number of sequences. For example, the Linux kernel contains 519,168 unique comment sentences. Pairwise comparison requires us to compare on the order of *100 billion* (134,767,706,112) pairs of word sequences to check if we can find rPairs from them. This is already the number after we filter out sequences that are too short or too long to as described later in Chapter 3.3. We ran the experiment on an Intel Core 2 Duo 3.06 HZ machine, and the pairwise comparison does not finish in one day.

To speed up the process, we want to reduce the number of pairwise comparisons. Our intuition is that there is no need to compare two sentences that do not share a single word. Therefore, we group sequences into *clusters*, one cluster for each word, where each cluster contains all the sequences that contain the word. We do not build clusters for words in the stopword list, which are words that appear frequently in English and software such as 'a', 'an', 'the', 'that', 'this', etc. Sharing only these non-essential words does not increase the similarity of the context for

discovering rPairs. We then conduct pairwise comparisons within each cluster. Since each cluster contains much fewer number of word sequences, this approach can significantly reduce the number of pairwise comparisons. For example, this step speeds up the analysis process for the Linux kernel by over 1,000 times: all the comments are divided into 123,404 clusters, and the total number of pairwise comparisons has been reduced to 90,483,147, which translates to only one hour on the same machine.

3.3 Extracting Semantically Related Word Pairs

The main step of the extraction process is to calculate the similarity between two word sequences and extract the corresponding word pairs if the similarity is higher than a given *threshold*. Since sequences are not always lined up from the first word, e.g., <`must`, `be`, `called`, `with`, `interrupts`, `disabled`> and <`it`, `must`, `be`, `called`, `with`, `interrupts`, `off`>, we apply the Longest Common Subsequence (LCS) algorithm to find the longest overlapping subsequences (not necessarily continuous) between two sequences.

We define the similarity measure as

$$SimilarityMeasure = \frac{\text{Number of Common Words in the Two Sequences}}{\text{Total Number of Words in the Shorter Sequence}}$$

If the similarity measure of a pair of word sequences is greater than or equal to the *threshold* (whose default value is *0.7* for the comment-comment context) and not 1 (meaning that the two sequences are identical), we extract rPairs from the differences between the two subsequences.

Our technique can find semantically related phrases, not only semantically related words. For example, from the sequences <`get`, `a`, `nodes's`, `parent` > and <`get`, `a`, `nodes's`, `left`, `child` >, we can find that the longest common subsequence of these two sequences is <`get`, `a`, `nodes's` >, and that phrases/words (*parent*, *left child*) are semantically related, because the $SimilarityMeasure$ is 0.75, which is greater than the default threshold.

In addition, we can find more than one rPair from two sequences. For example, from the sequences <`an`, `iovec`, `to`, `store`, `the`, `headers`, `sent`, `before`, `the`, `file`> and <`an`, `iovec`, `to`, `store`, `the`, `trailer`, `sent`, `after`, `the`, `file`>, we can infer two rPairs (*header*, *trailer*) and (*before*, *after*).

In addition to the *threshold*, three additional parameters are used to control the rPair extraction process: *shortest*, *longest*, and *gap*. Our technique only analyzes word sequences whose length is greater than or equal to *shortest* and less than or equal to *longest*, where sequence length is defined as the number of words in a sequence. Our technique only performs pairwise comparisons between two sequences whose length difference is *gap* or less.

3.4 Refining Semantically Related Word Pairs

We finally refine the detected rPairs. First, the rPairs in such format, (<*W1, W2*>, <*W3, W4*>) is separated into two rPairs, (*W1, W3*) and (*W2, W4*). Then we remove rPairs that contain words in the stopword list, e.g., (*a*, *the*); and we use stemming [1] to remove word pairs with the same roots, e.g., (*call*, *called*). Stemming is not perfect, e.g., Porter's stemmer makes mistakes such as stemming 'adding' to 'ad' instead of 'add'. However, it is widely used and works well for our experiments. We would like to experiment with other stemmers in the future.

In addition, since the same rPairs may be discovered from multiple pairs of sequences, we merge the word pairs as one rPair. For example, we can learn that (*interrupt*, *irq*) is an rPair from the two relevant comments in Table 3.1, as well as the two sequences <`were`, `called`, `from`, `interrupt`, `handlers`> and <`called`, `from`, `irq`, `handlers`>. We consider it as one rPair only, and increase the support for this rPair. The support is not used in our basic extraction technique, but it is used to rank the rPairs as described in Chapter 4.2.

Lastly, we normalize words to their base forms. For example, we normalize the rPair (*called, invoked*) to (*call*, *invoke*), and normalize the rPair (*threads*, *daemons*) to (*thread*, *daemon*). A typical stemmer is inappropriate for this normalization step, because a stemmer will revert words to their stems (e.g., *invoked* to *invok*), most of which are not words. In addition, stemming can cause inaccuracies as we discussed earlier regarding Porter's stemmer. Therefore, we build a reversely mapped dictionary that can return the base form of a word, given the derived form (e.g., past participles and plural nouns) of the word. We extract all base forms of English words and their derived forms from an English dictionary and build the reversely mapped dictionary. We normalize an rPair only if both words can be normalized. We require that both words can be normalized based on our observations. For example, the rPair (*disabled, off*) should not be normalized to (*disable, off*) because *disabled* and *off* are two semantically related adjectives, but the verb *disable* (the base form of *disabled*) is not a synonym of *off*.

We introduce *transitive rPairs*. If (W1, W2) and (W1, W3) are rPairs, (W2, W3) is a transitive rPair that requires one transition. If (W2, W4) is also an rPair, then (W3, W4) is a transitive rPair after two transitions. Considering transitive rPairs increases recall but reduces precision; our evaluation uses no transitive rPairs unless stated otherwise.

[1] http://tartarus.org/ martin/PorterStemmer/

Chapter 4

Improved Version of rPair Extraction

We propose two techniques to improve the rPair extraction accuracy: (1) we design a better similarity measure; and (2) we use an effective ranking function to rank the rPair results so that we can achieve a higher accuracy for the top ranked rPairs, i.e., top 10, top 30, and top 50 rPairs. Chapter 4.1 introduces how we use Inverse Document Frequency (*idf*) to redefine the *Similarity Measure* in Chapter 3.3. Chapter 4.2 briefly describes the ranking function we use.

4.1 An Improved Similarity Measure

In this section, we briefly describe how we adopt the *idf* technique to define a better similarity measure.

4.1.1 The *idf* Weight

By observing the results from our basic rPair extraction experiments, we notice that many false positives are introduced because many words in the shared context are less meaningful, e.g., the words such as 'both', 'via', and 'additional' are less meaningful than the words such as 'irq', 'disable', 'kernel', etc. Since the quality of the shared context could affect the accuracy of the inferred rPair, we give different weight values to different words to improve our similarity measure. In our experiment, we leverage the *idf* technique to assign different scores to every word. Therefore, those important and unique words are distinguished from those common and less meaningful words.

The *idf* is the inverse document frequency, and it is a widely-used metric to reflect how important a word is in a document or a collection of words. Typically, $idf(t, D)$ is defined as

$$idf(t, D) = \log_{base} \frac{|D|}{|d \in D : t \in d|}$$

where *D* represents all documents. The *idf* score ranges from *0* to $log_{base}D$. In our experiment, we normalize the *idf* as

$$normalized\_idf(t, D) = \frac{e^{idf(t,D)}}{1 + e^{idf(t,D)}}$$

to make it within $[0, 1)$ so that *idf* score is consistent across projects regardless of different number of methods in different projects.

4.1.2 The New Similarity Measure Definition

Based on the *normalized_idf* scores, we redefine the similarity measure as the following formula. To distinguish it from the similarity measure defined in Chapter 3.3, this improved similarity measure is referred to as the *new similarity measure*. In the following formula, s_1 and s_2 are two sentences, s_c represents the common part between s_1 and s_2, n_s is the number of words in sentence s, and S represents all the sentences.

$$NewSimilarityMeasure = \frac{2 \times \sum_{normalized\_idf}(s_c)}{\sum_{normalized\_idf}(s_1) + \sum_{normalized\_idf}(s_2)}$$

where

$$\sum_{normalized\_idf}(s) = \sum_{i=1}^{n_s} normalized\_idf(t_i, S)$$

4.2 The Ranking Function

Our goal is to design a ranking function to further improve the accuracy of our rPair extraction results. Two factors can be indicative of the correctness of rPairs: (1) the new similarity measure, and (2) the number of contexts in which one rPair can be learned, referred to as *support*. If the similarity measure of a rPair is higher, it is likely that it is a correct rPair. Similarly, if a rPair can be inferred from multiple contexts, it is likely that it is a correct rPair. Therefore, we combine the support and the similarity measure for effective ranking.

One common way to perform ranking based on multiple metrics is multi-objective ranking. We applied the NSGA-II algorithm [11], which is a popular multi-objective ranking algorithm based on dominance, and find that this ranking algorithm is ineffective in prioritizing correct rPairs.

Another common way to combine two factors to form a ranking function (if the two factors both have positive values), is to multiply these two factors. However, in our experiment, we observe that although the support plays an important role in ranking, the support and the similarity measure are in different ranges, e.g., the support of rPairs in the Linux kernel can be greater than 2,000, while the similarity measure is always between 0 and 1. If we simply use the product of the similarity measure and the *support* as the ranking function, the ranking function will be overpowered by the *support*. Therefore, we use a more balanced way of combining the *NewSimilarityMeasure* and the *Support*, by taking the logarithm of the support. Furthermore, we normalize the *RankingFunction* value with the logistic function.

We choose not to normalize the support score to $[0, 1)$; instead, we normalize the *RankingFunction* values to be consistent across the projects. We make this decision based on the observation that support should have a significant contribution in determining the correctness of rPairs, but a normalized support (e.g., using a logistic function) has limited such contribution. For example, the *normalized_support* of *support=5* has a limited difference from the *normalized_support* of *support* $>$ *5*, which makes the support contribute little to the ranking.

Our ranking function is:

$$RankingFunction = \begin{cases} \log_{base} Support \times AVG_{Similarity} & if\ Support > base \\ AVG_{Similarity} & if\ Support \leq base \end{cases}$$

where

$$AVG_{Similarity} = \frac{\sum_{i=1}^{Support} NewSimilarityMeasure}{Support}.$$

And the logistic function is:

$$normalized\_RankingFunction = \frac{e^{RankingFunction}}{1 + e^{RankingFunction}}.$$

While it is possible to use advanced ranking functions to further improve the accuracy, our simple ranking algorithm improves the rPair extraction accuracy (Chapter 6.4).

4.3 Studying Cross-Project rPairs

Some rPairs can be extracted from more than one project, which we call cross-project rPairs. For example, one rPair (*dev, device*) which appears in Linux, NetBSD and OpenBSD is one

cross-project rPair among the three operating system projects. The rPair (*dev, device*) is inferred from the Linux kernel comments "`disable cir logical` **dev**" and "`disable cir logical` **device**", from the NetBSD comments "`graphics` **dev** `is open`" and "`graphics` **device** `is open exclusive use`" and from the OpenBSD comments "`scsi` **dev** `clear operation`" and "`scsi` **device** `clear operation`".

Cross-project rPairs have many benefits. First, cross-project results can benefit other software from the same type of software which have not been analyzed yet. Second, cross-project rPairs are expected to have a higher accuracy; therefore whether an rPair is a cross-project rPair may be used to improve the ranking results.

In addition, we rank the cross-project rPairs. Specifically, we combine the average of the similarity measures and the sum of the supports from all projects the same way as in the ranking function in Chapter 4.2. We briefly describe how we conduct cross-project study in Chapter 5.5 and detailed cross-project rPair results with ranking are shown in Chapter 6.5.

Chapter 5

Experimental Methods

We evaluate our technique on nine open source projects (Table 5.1). Because method names are typically much shorter than comment sentences, we use different parameters for the comment-comment, code-code, and comment-code comparisons. For comment-comment comparisons, the parameter configuration is *shortest*=4, *longest*=10, *gap*=3, and *threshold*=0.7; for code-code comparisons, the parameter configuration is *shortest*=2, *longest*=4, *gap*=0, and *threshold*=0.5; and for comment-code comparisons, the parameter configuration is *shortest*=2, *longest*=6, *gap*=1, and *threshold*=0.6.

We perform five sets of evaluation experiments.

5.1 Experiment: rPair Extraction Accuracy and Comparison with WordNet and a Dictionary

We randomly sample 300 rPairs from all the rPairs generated for each project—100 rPairs extracted from the comment-comment context, 100 from the code-code context, and 100 from the comment-code context. We then manually read these rPairs and the corresponding word sequences to verify if the rPairs are correct rPairs. If fewer than 100 rPairs are extracted from one type of context in a code base, we manually verify all of the rPairs learned from that context in that code base. The *accuracy* is measured as the number of correct rPairs in a sample over the total number of rPairs in the sample. We further classify the correct rPairs into five categories—*synonym*, *related*, *antonym*, *near antonym*, or *identifier*, whose definition and examples are shown in Chapter 6.1. To reduce subjectivity, two people verify these results. In addi-

Table 5.1: Evaluated Software. LOComment is lines of comments. *The versions of iReport, jBidWatcher, javaHMO, jajuk are the same as [50]. The dates when NetBSD and OpenBSD were checked out from the version control systems are shown instead of the version numbers.

Software	**Source & Version**	**Description**	**LOC**	**LO-Comment**	**Lang--uage**
Linux	[61] 3.3	*The Linux kernel* Operating System	9,823,623	2,135,655	C
OpenBSD	[63] Feb2012	*OpenBSD* Operating System	2,029,168	487,623	C
NetBSD	[62] Nov2008	*NetBSD* Operating System	3,003,072	959,409	C
HTTPD	[58] 2.2.21	*Apache HTTPD Server* Web Server	231,526	70,229	C
Collections	[57] 3.2.1	*Apache Commons Collections* Libraries and Utilities	55,398	40,994	Java
iReport	[3] 1.2.2*	*iReport* Report Generator	74,506	18,614	Java
jBidWatcher	[2] 1.0pre6*	*jBidWatcher* eBay Auction Monitor	23,052	5,596	Java
javaHMO	[1] 2.4*	*javaHMO* Media Server	25,988	7,784	Java
jajuk	[4] 1.2*	*jajuk* Music Player	30,679	13,545	Java

tion, we check how many rPairs cannot be found in WordNet [44], an English dictionary [35] or a computer specific dictionary [10].

5.2 Experiment: Search-Related Evaluation

Previous work [50] builds a code search tool that expands search queries with alternative words learned from verb-DO pairs. For example, when developers search for "add textfield" in iReport, the tool will suggest words including *element*, *keyword*, and *token* for developers to select from to expand the initial query to queries such as *add element*, *add keyword*, *add token*, etc. These words are direct objects (DOs) that appear together with the verb *add* in iReport. To evaluate the technique, they manually identify the methods related to the concern "add textfield" in iReport,

referred to as *function gold set*, and check if such query expansions can improve the search effectiveness.

We perform two sets of search-related evaluation experiments. Firstly, we mimic the search process exhaustively by replacing the words in the search queries with rPairs mined by our approach, and compare our search results with those from SWUM. For example, for search task "add textfield", we first mimic the search process by searching with two queries "*add*textfiled*" and "*textfield*add*", because it is typical for users to reorder the verb and the noun to retrieve more search results. Second, we replace the words in the search queries, e.g., "add" and "textfield", with alternative words. To locate as many search results as possible, two words in the search query can be replaced together to form a new search query, for example, *"*new*reportpanel*"* is one valid search query if 'new' is alternative to 'add' and 'reportpanel' is alternative to 'textfield'. . For the search tasks with more than two words, such as *"Gather Music Files"*, we construct initial search queries by *"*verb*second noun*"*, such as *"*gather*file*"*, to locate more matches, because the query *"*gather*music*file*"* returns nothing in the project . We conducted the search using Eclipses' default search focusing on method declarations and constructors.

Secondly, since our technique is a building block for search tools, we compare the precision and recall of our rPair extraction results with the rPair extraction results of the previous work [50, 51, 16], on the *rPair gold set* inferred from the same search tasks [49] used by the previous work. For example, their function gold set for the search task "add auction" in jBidWatcher includes the following two methods "`AuctionsManager.newAuctionEntry(String)`" and "`AuctionServer.register- Auction(AuctionEntry)`", which means that when developers search for "add auction", these two methods should be matched. A keyword-based search for "add auction" in source code files will not find these methods. To locate them, we need to expand the query to "new auction" and "register auction". Therefore, we add two rPairs, (*add, new*) and (*add, register*), to our rPair gold set for the query word *add*. Note that the rPairs are added based on the function gold set. For example, the pair (*add, insert*) is not in our rPair gold set, because according to the function gold set, we do not need the word *insert* to locate the methods related to "add auction" in jBidWatcher. Since only eight of the nine search tasks from the previous work [50] require query expansion, we generate the rPair gold set for the eight search tasks.

For a fair comparison, we tune the previous technique to achieve the best performance, i.e., the highest recall, since it is harder to guess the words used in code, than to cross off false positives. First, we compare against their *latest and improved version* [16] (denoted by SWUM). Since the improved version analyzes only code but not comments, we can only compare our code-code analysis against their approach. If we add our comment-comment and comment-code analysis, our approach could find more rPairs as discussed in Chapter 6.2 and Chapter 2. Second, we relax one restriction of the SWUM technique to help it find rPairs that it may miss otherwise.

For example, for the query "*load movie*", the SWUM technique would suggest verbs that appear together with *movie*, which do not include *start*, because *start* does not appear together with *movie*. If the user decides to expand the query with the suggested words, the SWUM technique would suggest new words based on the new query. Therefore, whether *start* will eventually be suggested is uncertain. We relax this restriction so that the SWUM technique can find (*load*, *start*) as an rPair if *load* and *start* appear together with some DO, not necessarily *movie*.

In the first search-related experiment, based on the function gold set, we measure the *recall* as the number of methods in the gold set that a technique can discover over the total number of methods in the gold set. The *precision* is the number of methods in the gold set that a technique can discover over the total number of methods discovered by the technique with the expanded search queries including the original ones, such as "*add*textfield*" and "*textfield*add*" in the previous example. Similarity, in the second search-related experiment, based on the rPair gold set, we measure the *recall* as the number of rPairs in the gold set that a technique can discover over the total number of rPairs in the gold set. The *precision* is the number of rPairs in the gold set that a technique can discover over the total number of rPairs discovered by the technique that contain the original query word in the gold set (e.g., *add* and *load* in the previous examples).

5.3 Experiment: Sensitivity Evaluation

To understand how the *threshold* affects the performance of the proposed technique, ideally we want to vary the threshold, regenerate rPairs, and measure the precision and recall on a random sample of the rPairs. However, as the rPairs generated will be different with different threshold values, this evaluation approach requires a significant amount of effort on manually verifying the rPairs in the random samples. Therefore, as an approximation, we use the same random samples from the rPairs generated with our default threshold values (referred to as *default samples*), and measure the *recall* as the portion of the correct rPairs in a default sample that can be identified by our technique with a new threshold. The *precision* is the number of correct rPairs in the default sample that our technique can discover over the total number of rPairs in the default sample that our technique can discover.

5.4 Experiment: Ranking Evaluation

To evaluate whether the ranking function helps achieve a higher accuracy, we apply the ranking algorithm on the combined set of rPairs extracted from all three types of contexts from the nine

projects. We manually check the rPair accuracy of three ranking slots, i.e., top 10, top 30 and top 50, and compare the accuracy of these ranking slots with the accuracy from our basic extraction without ranking.

For each project, we apply the ranking algorithm on the rPair from the combined set of all the rPairs from the three categories of contexts (comment-comment, etc.). We choose *base*=10 (both in the ranking function and *idf(t, D)*). The accuracy results, which are verified by two people individually, are shown in Chapter 6.4.

5.5 Experiment: Cross-Project rPairs Study

Studying the overlapping rPairs across projects has many benefits, as discussed in Chapter 4.3. While our technique can be used to find overlapping rPairs among any projects, two projects of different types may not share any rPairs, e.g., jajuk and the Linux kernel share no rPairs. These two projects do not have similar functionality: jajuk is a media player while the Linux kernel is an operating system. Therefore, we focus on studying the overlapping rPairs among projects of the same type.

We experiment with two sets of projects of similar functionality. One set is two media players, i.e., jajuk and javaHMO, and the other set is three operating system projects, which are Linux, NetBSD and OpenBSD. For each set of projects, we conduct the cross-project study on the combined set of rPairs extracted from all three types of contexts and use the ranking algorithm described in Chapter 4.2 to rank the cross-project rPairs. In the future, we can apply our technique on other types of software, e.g., web browser projects such as Chrome, Firefox, etc.

5.6 Threats to Validity and Limitations

The search gold set and rPair gold set (introduced in Chapter 5) may favor a certain technique. To minimize this threat, we evaluate our technique on the same search gold set used by Shepherd et al. [50] as we compare against their technique. This is unlikely to favor our technique. In addition, two authors confirm the rPair gold set to reduce subjectivity.

If a code base contains no comments and the methods are poorly named, our technique may be less effective. However, given that modern software often contains a large amount of comments [55] and meaningful identifiers, our technique should be applicable to a large body of software. A large amount of commented code may affect the performance. First, it can add extra comments to the analysis, which are actually code. Second, analyzing code statements as

sentences may produce rPairs with lower accuracy because it is common for code statements to have many words in common. In the future, we can exclude commented code from our analysis to address this issue.

Our current implementation cannot tell if an rPair is *synonym*, *related*, *antonym*, *near antonym*, or *identifier*. Although all categories are useful for code search, it would be beneficial to distinguish these categories. In the future, we may leverage etymology to classify rPairs into the categories automatically.

Chapter 6

Evaluation Results

6.1 rPair Extraction Results

Table 6.1 shows the overall rPair extraction results on the nine evaluated code bases from the three types of contexts: comment-comment, code-code, and comment-code. We show the margin of error with 95% confidence level except for comment-comment of jBidWatcher and comment-code of HTTPD, iReport, and jajuk, of which we verify all extracted rPairs. We have two people manually verify the correctness of the rPairs, discuss the disagreements to reach consensus and report the disagreements statistics accordingly. In total, the two people have disagreements on 83 out of the 2269 verified rPairs. We calculate the Cohen's kappa value, which is a well-known statistical measure of inter-rater agreements, based on 6 categories of rPairs (Synonym, Related, Antonym, Near Antonym, Identifier, Incorrect). The Cohen's kappa value is 0.8668, which indicates "almost perfect agreement" according to [28].

We can see that the accuracy of the comment-comment context is the highest (30.0–84.0%) among the three contexts with an average of 56.9% (not shown in the Table), which is expected because comment sentences are generally longer than method names, which provides longer context for learning correct rPairs. In contrast, we learn fewer rPairs from the comment-code context, due to the disparity between comments and method names. However, the comment-code context does help us learn meaningful correct rPairs such as (*initialize*, *setup*) and (*alloc, add*), whose contexts are shown in Table 3.1.

Column 'Not in Dic or WordNet' shows the number of rPairs that cannot be found in either WordNet [44], an English dictionary and thesaurus [35] or an computer science specific dictionary [10]. These words and phrases are semantically related in software, but are not semantically

Table 6.1: rPair Extraction Results. Dic denotes Merriam-Webster English Dictionary and Thesaurus [35] and the computer specific dictionary [10]. The margin of error is calculated with 95% confidence level, which is to express the random sampling error in the extraction accuracy. The average accuracy of comment-comment is 56.9%, 33.3% for code-code and 16.5% for comment-code.

Software	**rPairs**	**Sample Size**	**Correct rPairs**	**Accuracy**	**#*Not* in Dic or WordNet**
Comment-Comment					
Linux	108,571	100	47	47.0±9.8%	36
HTTPD	1,428	100	47	47.0±9.5%	44
Collections	469	100	74	74.0±8.7%	72
iReport	878	100	84	84.0±9.2%	80
jBidWatcher	111	111	71	64.0%	63
javaHMO	144	100	56	56.0±5.4%	50
jajuk	203	100	69	69.0±7.0%	64
NetBSD	36,485	100	30	30.0±9.8%	30
OpenBSD	27,362	100	40	40.0±9.8 %	40
Code-Code					
Linux	606,432	100	25	25.0±9.8%	25
HTTPD	1,727	100	25	25.0±9.5%	24
Collections	3,162	100	41	41.0±9.7%	37
iReport	1,849	100	47	47.0±9.5%	47
jBidWatcher	1,428	100	42	42.0±9.5%	42
javaHMO	685	100	35	35.0±9.1%	35
jajuk	746	100	48	48.0±9.1%	47
NetBSD	354,680	100	20	20.0±9.8%	20
OpenBSD	223,323	100	17	17.0±9.8%	17
Comment-Code					
Linux	10,633	100	25	25.0±9.8%	25
HTTPD	43	43	12	27.9%	12
Collections	5	5	0	0	0
iReport	4	4	4	100%	4
jBidWatcher	0	0	0	0	0
javaHMO	0	0	0	0	0
jajuk	6	6	4	66.7%	4
NetBSD	1,169	100	7	7.0±9.38%	7
OpenBSD	703	100	7	7.0±9.08%	7

Table 6.2: rPairs Extraction Results. This table shows the percentage of correct rPairs distributed among five categories for nine projects. Majority of the correct rPairs (42.5-100%) belong to the 'Related' category.

Software	***Synonym***	***Related***	***Antonym-***	***Near Antonym***	***Identifier***
Comment-Comment					
Linux	2.1%	42.6%	2.1%	4.2%	49.0%
HTTPD	2.1%	51.0%	12.8%	2.1%	32.0%
Collections	0	78.4%	5.4%	4.1%	12.1%
iReport	0	50.0%	8.3%	1.2%	40.5%
jBidWatcher	0	66.2%	11.3%	16.9%	5.6%
javaHMO	1.8%	50.0%	7.1%	7.1%	34.0%
jajuk	5.8%	72.3%	8.7%	7.2%	0
NetBSD	0	63.3%	6.7%	0	30.0%
OpenBSD	2.5%	42.5%	2.5%	2.5%	50.0%
Code-Code					
Linux	4.0%	84.0%	4.0%	8.0%	0
HTTPD	4.0%	72.0%	12.0%	12.0%	0
Collections	4.9%	82.9%	7.3%	4.9%	0
iReport	2.1%	93.6%	2.1%	2.1%	0
jBidWatcher	2.4%	85.7%	2.4%	9.5%	0
javaHMO	0	94.2%	2.9%	0	2.9%
jajuk	0	89.6%	6.3%	4.1%	0
NetBSD	0	100%	0	0	0
OpenBSD	0	100%	0	0	0
Comment-Code					
Linux	0	88.0%	0	12.0%	0
HTTPD	8.3%	91.7%	0	0	0
Collections	0	0	0	0	0
iReport	0	100%	0	0	0
jBidWatcher	0	0	0	0	0
javaHMO	0	0	0	0	0
jajuk	0	100%	0	0	0
NetBSD	0	85.7%	0	14.3%	0
OpenBSD	0	85.7%	0	14.3%	0

related in English. This is very valuable because it is almost impossible for developers to guess all the semantically related words used in a given piece of software. Our results show that 711 out of the 756 (94.0%) correct rPairs in the nine projects cannot be found in either WordNet [44], an English dictionary [35] or a computer specific dictionary [10].

The breakdown of the correct rPairs into five categories is shown in Table 6.2. *Synonym* denotes words that have the same meanings in software (including abbreviations), e.g., (*call*, *invoke*) and (*interrupt*, *irq*). *Related* denotes words that are semantically related but not the same, e.g., (*size*, *capacity*) and (*file*, *directory*). *Antonym* denotes words that have opposite meanings, e.g., (*serialize*, *deserialize*) and (*before*, *after*). *Near Antonym* denotes words that have almost opposite meanings, e.g., (*header*, *trailer*). The full contexts of these rPair examples are shown in Table 3.1. *Identifier* denotes words that are semantically related code identifiers, such as method names, variable names, etc. For example, *makeFullMap()* and *makeEmptyMap()* are a pair of *identifier* rPairs, which are two function names from Collections. All five types of rPairs are useful for code search and other software engineering tasks.

False Positives. Despite the challenging nature of the task, our technique has reasonable accuracy. However, there is much space to further improve the accuracy. One main cause of false positives is that the shared context contains many common English words. For example, we mistakenly consider (*match*, *literal*) semantically related, from comments "`we have a match`", and "`we have a literal`". Another reason is that our design favors recall over precision; the threshold and the support (the number of contexts from which the rPairs can be learned) are set low, and the gap (the length difference between two sequences compared) allowed is high. Despite the false positives, our techniques is valuable, because it is much easier for developers to cross off false positives than to guess the possible semantically related words used in software.

To reduce false positives, we rank rPairs according to the importance of the words in the shared context (e.g., *idf* scores) and the support; the results are in Chapter 6.4. In addition, we could leverage NLP techniques to generate the semantic paths [33] to infer rPairs more precisely. At the cost of lower recall, users can increase the threshold and decrease the gap to improve the precision.

6.2 Search-Related Results

Table 6.3 shows the search-related results on the methods gold set. Column 'Initial Search Query' shows the initial search queries established based on the search task. We conduct the search experiments by expanding the initial search queries by replacing the words in the queries with the words mined by two techniques. For example, for the search task "add auction" in jBidWatcher,

Table 6.3: Search-Related results based on function gold set. We search by "*verb*noun*", "*noun*verb*" and combinations of alternative words for 'verb' and 'noun'. CTX is our context-based technique, CTX_T is CTX with transitive rPairs, and SWUM denotes the previous work by [16]. The function gold set is from [49].

Search Task	**Initial Search Query**	**Precision**			**Recall**		
		CTX	SWUM	CTX_T	CTX	SWUM	CTX_T
iReport							
"Add Textfield"	*add*textfield* *textfield*add*	14.3%	0	5.3%	40.0%	0	60%
"Compile Report"	*compile*report *report*compile*	4%	15.8%	0.17%	25%	37.5%	87.5%
javaHMO							
"Gather Music Files"	*gather*file* *file*gather*	28.6%	0.23%	0.5%	50%	50%	75%
"Load Movie Listing"	*load*listing* *listing*load*	0	0	0	0	0	0
jBidWatcher							
"Add Auction"	*add*auction* *auction*add*	3.7%	0.82%	0.94%	100%	100%	100%
"Save Auction"	*save*auction* *auction*save*	1.61%	0.52%	0.66%	33.3%	66.7%	77.8%
"Set Snipe Price"	*set*price* *price*set*	0	0	0/0	0	0	0/0
jajuk							
"Play Song"	*play*song* *song*play*	0	0	0/0	0	0	0/0

Table 6.4: Search-Related results on rPairs gold set. CTX is our context-base technique, CTX_T is CTX with transitive rPairs, and SWUM denotes the previous work by [16].

Search Task	**rPairs in Gold Set**	**Precision**			**Recall**		
		CTX	SWUM	CTX_T	CTX	SWUM	CTX_T
iReport							
"Add Textfield"	`add->new,drop`	3.7%	0	0.3%	50.0%	0	50.0%
	`TextField`	0	0	0	0	0	0
	`->ReportPanel`						
"Compile Report"	`report->directory`	0	0	0.3%	0	0	100%
javaHMO							
"Gather Music Files"	`gather->find`	0	0	1.1%	0	0	100%
	`file`	3.3%	0.4%	0.6%	100%	100%	100%
	`->directory`						
"Load Movie Listings"	`listing`	0	0	0	0	0	0
	`->container`						
	`load->start`	20.0%	5.3%	1.1%	100%	100%	100%
jBidWatcher							
"Add Auction"	`add->do,new,`	5.5%	2.3%	1.0%	100%	66.7%	100%
	`register`						
	`auction->entry`	1.8%	0.3%	0.4%	100%	100%	100%
"Save Auction"	`save->preserve`	0	1.6%	0.3%	0	33.3%	33.3%
	`backup,do`						
"Set Snipe Price"	`price->currency`	0	0	0/0	0	0	0/0
jajuk							
"Play Song"	`song->file,`	0	0	0/0	0	0	0/0
	`playlist`						
	`play->launch`	0	0	0/0	0	0	0/0

we perform two initial search queries—"*add*auction*" and "*auction*add*" to mimic typical search queries from developers. We expand the query word *add* with its semantically related words, *new*, *register*, and *do*, to locate the relevant methods. Column 'Precision' shows the percentage of the functions the two techniques find correctly and Column 'Recall' shows the percentage of the functions in the gold set which the two techniques can find. Column 'CTX' is our context-based technique, column 'CTX_T' is CTX with transitive rPairs, and SWUM denotes the previous work [16, 50].

Table 6.4 shows the search-related results on the rPairs gold set. Column 'rPairs in gold set' shows all the rPairs in our *rPair gold set*, which can help expand the search queries to find the

relevant methods. Column 'Precision' shows the percentage of the rPairs the two techniques find correctly and Column 'Recall' shows the percentage of the rPairs in the gold set which the two techniques can find.

As stated in Chapter 5, we tune the SWUM technique to reach its best performance. Without the tuning, the SWUM technique would potentially miss three additional rPairs, (*load*, *start*), (*add*, *do*), and (*file*, *directory*). Thus SWUM will miss some functions in the function gold set, such as *FileGatherer.gatherDirectory(File, String, FileFilter, Z)* for the search task "gather music files".

Overall, our context-based approach (CTX) outperforms the SWUM approach on the two sets of search-related experiments. In the search-related experiment on function gold set, for three search tasks, both techniques have zero recall. For three out of the five remaining tasks, our context-based approach has higher recall and precision or same recall with higher precision. Furthermore, our technique with transitive pairs (at most two transitions allowed CTX_T) can achieve higher recall for all the remaining five tasks and higher precision for four out of the five tasks than SWUM. SWUM has higher precision and higher recall than our technique on the search task "compile report" because our technique fails to discover *'translated compile directory'* is related to *'report'*. Thus SWUM can locate one more function *MainFrame.getTranslatedCompileDirectory()*. One caveat is that users may not use the full rPairs to expand the search queries, however we mimic the scenario maximally by leveraging the entire results.

In the search-related experiment on rPairs gold set, for three search tasks, both techniques have a zero recall. For four out of the five remaining tasks, our context-based approach has higher recall and precision or same recall with higher precision. For example, our technique can find the rPair (*add*, *new*) in iReport, but the previous approach will miss it because NLP tools trained from general English text will not consider *new* a verb as discussed in Introduction. One caveat is that using these semantically related words to expand queries may locate more irrelevant method names. However, recent techniques [18, 19] may be leveraged to restrict the search context and scope to address this issue.

For the rPairs that both techniques can find, our technique (CTX) has a higher precision (by a factor of 2.4–8.3). This is because we use similarity measures to filter out irrelevant pairs, and we do not consider return types or parameters to minimize the dependency on naming convention and code structure related heuristics. Although these design choices may make our technique discover fewer rPairs, they did not cause our technique to miss any rPairs in the search-related rPair gold set that the SWUM technique can find.

Although our technique (CTX) has a higher precision than the SWUM technique, the precision of both techniques is relatively low, because (1) this is an inherently challenging task, and

(2) we only count the particular rPairs in our gold set as true positives, and consider other correct rPairs discovered by the techniques as false positives, which can be useful for other search queries nonetheless. We can use the techniques discussed in Chapter 6.1 to improve the precision.

The only case that the SWUM approach has a higher recall than our approach is for the rPair (*save*, *do*). The SWUM approach breaks the method name `DoSave` into two verbs, and generate two verb-DOs by combing the method name with the parameter name. Our technique does not attempt to break one method name into two sequences, therefore misses this rPair. However, our technique with transitive pairs (CTX_T) can find this rPair.

In addition, Table 6.4 shows that by considering transitive rPairs with at most two transitions allowed (CTX_T), we can find three additional rPairs in the gold set with lower precisions, including two rPairs that neither the SWUM technique nor our technique without transitive rPairs (CTX) can find—(*gather*, *find*) and (*report*, *directory*). If we analyze comments as well, then an additional rPair (*listing*, *container*) will be identified.

6.3 Sensitivity Results

To understand how a higher threshold affects the precision and recall on the comment-comment analysis, we first experiment with thresholds 0.8, and 0.9. We found that threshold 0.8 significantly reduces the number of identified rPairs (recalls are lower than 0.5 for all nine projects) with much higher precision (7.3–56.3% improvement) for all projects except for HTTPD (a small improvement of 0.6%), and threshold 0.9 finds zero rPairs in our default samples for iReport, HTTPD, and the Linux kernel, and one rPair for the rest 4 projects. Since we favor recall, we then evaluate a lower threshold 0.75, which, however, still gives us low recalls (less than 0.5 for all projects) with precisions lower than those with threshold 0.8. Since the number of words in a sentence is integers, thresholds between 0.7 and 0.75 are either equivalent to 0.7 or 0.75; therefore, there is no need to evaluate them. In summary, threshold 0.7 finds more rPairs with reasonable precisions; therefore, it was chosen as our default value. If higher precision is preferred and lower recall is acceptable, 0.8 is a good choice. We did not conduct the same experiment for the comment-code and code-code context because method names are generally short, e.g., two words, which means that only a small set of discrete similarity measure values are possible: 0 (no shared word between the two method names), 0.5 (one shared word), and 1 (two shared words). Threfore, 0.5 is the only reasonable threshold to set, indicating that threshold tuning is not meaningful.

Table 6.5: The accuracy of rPair extraction with ranking. Top 10 can achieve the highest accuracy of 60-100%, top 30 achieve the accuracy of 66.7-83.3% and top50 has the accuracy of 60-80%.

Software	**Total**	Ranking Slots			**Comment-Comment**
		Top 10	Top 30	Top 50	
HTTPD	3,053	60.0%	66.7%	66.0%	47.0%
Collections	3,535	100.0%	83.3%	70.0%	74.0%
Linux	196,272	90.0%	66.7%	67.0%	47.0%
iReport	7,474	90.0%	83.3%	80.0%	84.0%
jBidWatcher	1,537	80.0%	63.3%	68.0%	64.0%
javaHMO	822	70.0%	73.3%	62.0%	56.0%
jajuk	915	70.0%	73.3%	78.0%	69.0%
NetBSD	391,000	100%	73.3%	60.0%	30.0%
OpenBSD	250,897	100%	83.3%	60.0%	40.0%

6.4 Ranking Results

This chapter shows the accuracy of the ranked rPairs on the nine evaluated code bases. Table 6.5 shows the total number of rPairs for each software using the parameter configuration described in Chapter 5.4 and the accuracy in the three ranking slots—top 10, top 30 and top 50. The accuracy is defined as the portion of correct rPairs in the results. Table 6.5 also shows the rPair extraction accuracy from the comment-comment context on the nine code bases (copied from Table 6.1) for comparison. Note that the ranking results are performed on all three types of contexts combined, while the basic extraction results from the comment-comment context is the highest among the three types of contexts without ranking.

The top 10 rPairs have a much higher accuracy of 60.0-100% for all nine code bases (compared to the 30.0-84.0% from comment-comment without ranking, which has the highest accuracy among the three types of contexts). The top 30 rPairs achieve a higher accuracy of 63.3-83.3% for all nine projects except iReport and jBidWatcher. Even for iReport and jBidWatcher, the accuracy is marginally lower (84.0% to 83.3% for iReport and 64.0% to 63.3% for jBidWatcher). Lastly, the accuracy of the top 50 rPairs is also improved to 62.0-80.0% for all evaluated projects except Collections (from 74.0% to 70.0%) and iReport (from 84.0% to 80.0%).

Our results show that ranking can significantly improve the rPair extraction accuracy, and the accuracy improvement is bigger for large projects (i.e., the operating system code bases—Linux,

Table 6.6: Cross-project rPair Results.

Software	Type	Total	Top N	Accuracy	Examples
jajuk javaHMO	Media Player	8	8	87.5%	(load, clear) (item, file)
Linux NetBSD OpenBSD	Operating System	8,667	50	68%	(dev, device) (reg, register) (receive, transmit) (destination, source) (free, allocate)

NetBSD and OpenBSD). This is probably because the *support* for larger projects is significantly bigger, which can help highlight correct rPairs extracted from many contexts in comments and code. For example, *support* of rPairs from Linux can be as high as hundreds (i.e., support of rPair (*read, write*) is 507), which results in a higher accuracy improvement. *Support* of rPairs in small projects is much smaller. For example, the *support* of most rPairs in jajuk is less than 10. In this case, the support does not contribute much to the ranking function (since the *base* is 10 in our experiment setting). Therefore, for small projects i.e., jajuk, jBidWatcher and javaHMO, the rPairs are ranked based on the similarity measure. Since *support* plays an important role in determining the correctness of the rPairs, a hybrid ranking function which combines the *support* and *the similarity measure* is expected to have better ranking results. This explains why our ranking function has better performance on large projects.

6.5 Cross-Project rPair Results

Table 6.6 shows the cross-project rPair results from two sets of software. One set contains two media players—jajuk and javaHMO. The other set is three operating systems—Linux, NetBSD and OpenBSD. We show the total number of overlapping rPairs across projects of the same set (Column "Total"), the top number of rPairs that we manually verify (Column "Top N"), and the rPair accuracy from our manual verification. The last column shows some cross-project rPair examples.

We identify 8 cross-project rPairs from the two media players, and 8,667 cross-project rPairs from the three operating system projects. For the media players, the extraction accuracy is 87.5%, which is higher than the project-specific rPair extraction accuracy for both jajuk and javaHMO rPairs with ranking (the highest accuracy from the three ranking slots is 78% for jajuk and 73.3%

for javaHMO). For the operating system projects, the top 50 cross-project rPairs have a higher accuracy (68.0%) than that of the project-specific ranking results (the accuracy of the top 50 rPairs is 67.0% for Linux, 60.0% for NetBSD and 60.0% for OpenBSD). The results show that cross-project rPairs have a higher accuracy than project-specific rPairs; therefore, it is promising to improve the ranking performance by adding "cross-project" as a factor in the ranking algorithm.

We find that most of the correct rPairs in the cross-project results are general for that type of software. For example, the cross-project rPair (*dev, device*) found in all three operating system code bases is general for operating systems (i.e., *dev* is highly likely to mean *device* in operating system code), since operating systems need to manage devices. This rPair is more general than other project-specific rPairs, e.g., the rPair (*use_hcd_c_probe, use_hcd_sa_probe*) from Linux, which are the names of two related functions in Linux.

Cross-project rPairs may be leveraged (for code search, etc.) by projects of the same type that are not analyzed. For example, we may be confident that *dev* is the abbreviated form of *device* in operating systems that we have not analyzed, e.g., OpenSolaris and Windows. Therefore, we believe our cross-project rPairs can benefit other media players and operating system projects, for example, the cross-project rPair *(dev, device)* can supplement other operating system projects in which this rPair is not discovered from the comments and code of those projects.

Chapter 7

Conclusions

We design and evaluate a general technique to automatically discover semantically related words in software by leveraging the context of words in comments and code. The proposed technique identifies semantically related words with a reasonable accuracy in nine large and popular code bases written in C and Java. Our further evaluation against the state of art shows that our overall precision and recall in discovering semantically related word pairs is higher.

In addition, we propose a ranking algorithm and evaluate it on the nine projects. Our evaluation shows that the ranking algorithm can improve the accuracy of the rPair extraction. The cross-project rPairs study shows that we can discover general overlapping rPairs across multiple projects of similar functionality and cross-project rPairs are more likely to be accurate than project-specific rPairs.

This context-based approach and the rPairs discovered are valuable as they can benefit many software engineering tasks. Firstly, semantically related words from software can help improve the search effectiveness of keyword-based code search tools [16, 18, 19, 41, 42, 50] by expanding the search queries. Secondly, semantically related words can improve bug detection tools [37, 43, 55, 56, 67] by expanding the specified rules with synonyms for better software reliability. Thirdly, other software engineering tasks such as detecting duplicate bug reports and mining source code descriptions need to analyze natural-language artifacts (e.g., bug reports [6, 14, 31, 34, 39, 46, 54, 64] and mailing lists [38]). They can leverage semantically related words to improve their work.

In the future, we plan to use our technique to build comprehensive and accurate databases of semantically related words in software. In addition, we can analyze user manuals and other software documents to learn semantically related words. Furthermore, we may leverage etymology to classify rPairs into different categories, e.g., synonym and antonyms, automatically.

References

[1] JavaHMO. http://www.javahmo.sourceforge.net/, 2009.

[2] JBidWatcher. http://www.jbidwatcher.com/, 2011.

[3] iReport. http://jasperforge.org/projects/ireport, 2012.

[4] Jajuk. http://www.jajuk.info/index.php/Main_Page, 2012.

[5] Surafel Lemma Abebe and Paolo Tonella. Natural language parsing of program element names for concept extraction. In *Proceedings of the 2010 IEEE 18th International Conference on Program Comprehension*, ICPC '10, pages 156–159, Washington, DC, USA, 2010. IEEE Computer Society.

[6] John Anvik, Lyndon Hiew, and Gail C. Murphy. Who should fix this bug? In *Proceedings of the 28th international conference on Software engineering*, ICSE '06, pages 361–370, New York, NY, USA, 2006. ACM.

[7] Satanjeev Banerjee and Ted Pedersen. Extended gloss overlaps as a measure of semantic relatedness. In *Proceedings of the 18th international joint conference on Artificial intelligence*, IJCAI'03, pages 805–810, San Francisco, CA, USA, 2003. Morgan Kaufmann Publishers Inc.

[8] Dave Binkley, Matthew Hearn, and Dawn Lawrie. Improving identifier informativeness using part of speech information. In *Proceedings of the 8th Working Conference on Mining Software Repositories*, MSR '11, pages 203–206, New York, NY, USA, 2011. ACM.

[9] Raymond P.L. Buse and Thomas Zimmermann. Information needs for software development analytics. In *Proceedings of the 34th International Conference on Software Engineering*, June 2012.

[10] Computer Dictionary Online. Computer Science Specific Dictionary. `http://www.computer-dictionary-online.org`, 2013.

[11] K. Deb, A. Pratap, S. Agarwal, and T. Meyarivan. A fast and elitist multiobjective genetic algorithm: NSGA-II. *Trans. Evol. Comp*, 6(2):182–197, April 2002.

[12] Zachary P. Fry, David Shepherd, Emily Hill, Lori Pollock, and K. Vijay-Shanker. Analysing source code: Looking for useful verb-direct object pairs in all the right places. *IET Software*, 2008.

[13] 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, November 1987.

[14] Michael Gegick, Pete Rotella, and Tao Xie. Identifying security bug reports via text mining: An industrial case study. In Jim Whitehead and Thomas Zimmermann, editors, *MSR*, pages 11–20. IEEE, 2010.

[15] Scott Grant, Douglas Martin, James R Cordy, and David B Skillicorn. Contextualized semantic analysis of web services. In *International Symposium on Web Systems Evolution*, 2011.

[16] Emily Hill. *Integrating natural language and program structure information to improve software search and exploration*. PhD thesis, Newark, DE, USA, 2010. AAI3423409.

[17] Emily Hill, Zachary P. Fry, Haley Boyd, Giriprasad Sridhara, Yana Novikova, Lori Pollock, and K. Vijay-Shanker. AMAP: Automatically mining abbreviation expansions in programs to enhance software maintenance tools. In *Proceedings of the 2008 international working conference on Mining software repositories*, MSR '08, pages 79–88, New York, NY, USA, 2008. ACM.

[18] Emily Hill, Lori Pollock, and K. Vijay-Shanker. Automatically capturing source code context of NL-queries for software maintenance and reuse. In *Proceedings of the 31st International Conference on Software Engineering*, ICSE '09, pages 232–242, Washington, DC, USA, 2009. IEEE Computer Society.

[19] Emily Hill, Lori Pollock, and K. Vijay-Shanker. Improving source code search with natural language phrasal representations of method signatures. In *Proceedings of the 2011 26th IEEE/ACM International Conference on Automated Software Engineering*, ASE '11, pages 524–527, Washington, DC, USA, 2011. IEEE Computer Society.

[20] Emily Hill, Lori Pollock, and K. Vijay-Shanker. Investigating how to effectively combine static concern location techniques. In *Proceedings of the 3rd International Workshop on Search-Driven Development: Users, Infrastructure, Tools, and Evaluation*, SUITE '11, pages 37–40, New York, NY, USA, 2011. ACM.

[21] Abram Hindle, Earl T. Barr, Zhendong Su, Mark Gabel, and Premkumar Devanbu. On the naturalness of software. In *Proceedings of the 2012 International Conference on Software Engineering*, ICSE 2012, pages 837–847, Piscataway, NJ, USA, 2012. IEEE Press.

[22] Reid Holmes and Gail C. Murphy. Using structural context to recommend source code examples. In *Proceedings of the 27th international conference on Software engineering*, ICSE '05, pages 117–125, New York, NY, USA, 2005. ACM.

[23] Einar W. Host and Bjarte M. Ostvold. The programmer's lexicon, volume I: The verbs. In *Proceedings of the Seventh IEEE International Working Conference on Source Code Analysis and Manipulation*, SCAM '07, pages 193–202, Washington, DC, USA, 2007. IEEE Computer Society.

[24] Einar W. Host and Bjarte M. Ostvold. Software language engineering. pages 322–341, Berlin, Heidelberg, 2009. Springer-Verlag.

[25] Doug Janzen and Kris De Volder. Navigating and querying code without getting lost. AOSD, 2003.

[26] Jay J. Jiang and David W. Conrath. Semantic similarity based on corpus statistics and lexical taxonomy. *CoRR*, 1997.

[27] Andrew J. Ko, Htet Aung, and Brad A. Myers. Eliciting design requirements for maintenance-oriented IDEs: A detailed study of corrective and perfective maintenance tasks. In *Proceedings of the 27th international conference on Software engineering*, ICSE '05, pages 126–135, New York, NY, USA, 2005. ACM.

[28] J. Richard Landis and Gary G. Koch. The measurement of observer agreement for categorical data. *Biometrics*, 33(1):pp. 159–174, 1977.

[29] Dawn Lawrie and Dave Binkley. Expanding identifiers to normalize source code vocabulary. In *Proceedings of the 2011 27th IEEE International Conference on Software Maintenance*, ICSM '11, pages 113–122, Washington, DC, USA, 2011. IEEE Computer Society.

[30] Dawn Lawrie, Dave Binkley, and Christopher Morrell. Normalizing source code vocabulary. In *Proceedings of the 2010 17th Working Conference on Reverse Engineering*, WCRE '10, pages 3–12, Washington, DC, USA, 2010. IEEE Computer Society.

[31] Zhenmin Li, Lin Tan, Xuanhui Wang, Shan Lu, Yuanyuan Zhou, and Chengxiang Zhai. Have things changed now?: An empirical study of bug characteristics in modern open source software. In *Proceedings of the 1st workshop on Architectural and system support for improving software dependability*, ASID '06, pages 25–33, New York, NY, USA, 2006. ACM.

[32] Dekang Lin. An information-theoretic definition of similarity. In *Proceedings of the Fifteenth International Conference on Machine Learning*, ICML '98, pages 296–304, San Francisco, CA, USA, 1998. Morgan Kaufmann Publishers Inc.

[33] Dekang Lin and Patrick Pantel. Discovery of inference rules for question-answering. *Nat. Lang. Eng.*, 7(4):343–360, December 2001.

[34] Dominique Matter, Adrian Kuhn, and Oscar Nierstrasz. Assigning bug reports using a vocabulary-based expertise model of developers. In *Proceedings of the 2009 6th IEEE International Working Conference on Mining Software Repositories*, MSR '09, pages 131–140, Washington, DC, USA, 2009. IEEE Computer Society.

[35] Merriam-Webster. Merriam-Webster English Dictionary and Thesaurus. `http://www.merriam-webster.com`, 2012.

[36] Oracle Corporation. OpenSolaris. `http://hub.opensolaris.org/bin/view/Main/`, 2012.

[37] Rahul Pandita, Xusheng Xiao, Hao Zhong, Tao Xie, Stephen Oney, and Amit Paradkar. Inferring method specifications from natural language API descriptions. In *Proceedings of the 2012 International Conference on Software Engineering*, ICSE 2012, pages 815–825, 2012.

[38] Sebastiano Panichella, Jairo Aponte, Massimiliano Di Penta, Andrian Marcus, and Gerardo Canfora. Mining source code descriptions from developer communications. In *ICPC*, pages 63–72, 2012.

[39] Jin-Woo Park, Mu-Woong Lee, Jinhan Kim, Seung won Hwang, and Sunghun Kim. CosTriage: A cost-aware triage algorithm for bug reporting systems. In Wolfram Burgard and Dan Roth, editors, *AAAI*. AAAI Press, 2011.

[40] Denys Poshyvanyk and Andrian Marcus. Combining formal concept analysis with information retrieval for concept location in source code. In *ICPC*, 2007.

[41] Denys Poshyvanyk, Andrian Marcus, and Yubo Dong. JIRiSS - an eclipse plug-in for source code exploration. In *Proceedings of the 14th IEEE International Conference on Program Comprehension*, ICPC '06, pages 252–255, Washington, DC, USA, 2006. IEEE Computer Society.

[42] Denys Poshyvanyk, Maksym Petrenko, Andrian Marcus, Xinrong Xie, and Dapeng Liu. Source code exploration with google. In *Proceedings of the 22nd IEEE International Conference on Software Maintenance*, ICSM '06, pages 334–338, Washington, DC, USA, 2006. IEEE Computer Society.

[43] Michael Pradel, Severin Heiniger, and Thomas R. Gross. Static detection of brittle parameter typing. In *Proceedings of the 2012 International Symposium on Software Testing and Analysis*, ISSTA 2012, pages 265–275, New York, NY, USA, 2012. ACM.

[44] Princeton University. WordNet. `http://wordnet.princeton.edu`, 2012.

[45] Philip Resnik. Using information content to evaluate semantic similarity in a taxonomy. In *Proceedings of the 14th international joint conference on Artificial intelligence - Volume 1*, IJCAI'95, pages 448–453, San Francisco, CA, USA, 1995. Morgan Kaufmann Publishers Inc.

[46] Per Runeson, Magnus Alexandersson, and Oskar Nyholm. Detection of duplicate defect reports using natural language processing. In *Proceedings of the 29th international conference on Software Engineering*, ICSE '07, pages 499–510, Washington, DC, USA, 2007. IEEE Computer Society.

[47] Gerard Salton and Christopher Buckley. Readings in information retrieval. chapter Term-weighting approaches in automatic text retrieval, pages 323–328. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1997.

[48] Zachary M. Saul, Vladimir Filkov, Premkumar Devanbu, and Christian Bird. Recommending random walks. In *Proceedings of the the 6th joint meeting of the European software engineering conference and the ACM SIGSOFT symposium on The foundations of software engineering*, ESEC-FSE '07, pages 15–24, New York, NY, USA, 2007. ACM.

[49] David Shepherd. Action-oriented concerns. `http://www.eecis.udel.edu/~gibson/context/action_oriented_concerns.txt`, 2007.

[50] David Shepherd, Zachary P. Fry, Emily Hill, Lori Pollock, and K. Vijay-Shanker. Using natural language program analysis to locate and understand action-oriented concerns. In

Proceedings of the 6th international conference on Aspect-oriented software development, AOSD '07, pages 212–224, New York, NY, USA, 2007. ACM.

[51] David Shepherd, Lori Pollock, and K. Vijay-Shanker. Towards supporting on-demand virtual remodularization using program graphs. In *Proceedings of the 5th international conference on Aspect-oriented software development*, AOSD '06, pages 3–14, New York, NY, USA, 2006. ACM.

[52] Giriprasad Sridhara, Emily Hill, Divya Muppaneni, Lori Pollock, and K. Vijay-Shanker. Towards automatically generating summary comments for Java methods. In *ASE*, 2010.

[53] Giriprasad Sridhara, Emily Hill, Lori Pollock, and K. Vijay-Shanker. Identifying word relations in software: A comparative study of semantic similarity tools. In *Proceedings of the 2008 The 16th IEEE International Conference on Program Comprehension*, ICPC '08, pages 123–132, Washington, DC, USA, 2008. IEEE Computer Society.

[54] Chengnian Sun, David Lo, Xiaoyin Wang, Jing Jiang, and Siau-Cheng Khoo. A discriminative model approach for accurate duplicate bug report retrieval. In *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering - Volume 1*, ICSE '10, pages 45–54, New York, NY, USA, 2010. ACM.

[55] Lin Tan, Ding Yuan, Gopal Krishna, and Yuanyuan Zhou. /* iComment: Bugs or bad comments?*/. In *Proceedings of twenty-first ACM SIGOPS symposium on Operating systems principles*, SOSP '07, pages 145–158, New York, NY, USA, 2007. ACM.

[56] Lin Tan, Yuanyuan Zhou, and Yoann Padioleau. aComment: Mining annotations from comments and code to detect interrupt related concurrency bugs. In *Proceedings of the 33rd International Conference on Software Engineering*, ICSE '11, pages 11–20, New York, NY, USA, 2011. ACM.

[57] The Apache Foundation. Apache Commons Collections. `http://commons.apache.org/collections/`, 2012.

[58] The Apache Foundation. Apache HTTPD Server. `http://httpd.apache.org`, 2012.

[59] The Apache Software Foundation. Apache OpenNLP. `http://opennlp.apache.org`, 2010.

[60] The FreeBSD Foundation. FreeBSD. `http://www.freebsd.org`, 2012.

[61] The Linux Kernel Organization, Inc. The Linux Kernel. `http://www.kernel.org`, 2012.

[62] The NetBSD Foundation. NetBSD. `http://www.netbsd.org`, 2012.

[63] The OpenBSD Foundation. OpenBSD. `http://www.openbsd.org`, 2012.

[64] Xiaoyin Wang, Lu Zhang, Tao Xie, John Anvik, and Jiasu Sun. An approach to detecting duplicate bug reports using natural language and execution information. In *Proceedings of the 30th international conference on Software engineering*, ICSE '08, pages 461–470, New York, NY, USA, 2008. ACM.

[65] Zhibiao Wu and Martha Palmer. Verbs semantics and lexical selection. In *Proceedings of the 32nd annual meeting on Association for Computational Linguistics*, ACL '94, pages 133–138, Stroudsburg, PA, USA, 1994. Association for Computational Linguistics.

[66] Amy Moormann Zaremski and Jeannette M. Wing. Signature matching: A tool for using software libraries. *ACM Trans. Softw. Eng. Methodol.*, 4(2):146–170, April 1995.

[67] Hao Zhong, Lu Zhang, Tao Xie, and Hong Mei. Inferring resource specifications from natural language API documentation. In *Proceedings of the 2009 IEEE/ACM International Conference on Automated Software Engineering*, ASE '09, pages 307–318, Washington, DC, USA, 2009. IEEE Computer Society.