UC Irvine
UC Irvine Electronic Theses and Dissertations

Title
Beyond Similar Code: Leveraging Social Coding Websites

Permalink
https://escholarship.org/uc/item/825526n1

Author
Yang, Di

Publication Date
2019

Peer reviewed|Thesis/dissertation

eScholarship.org Powered by the California Digital Library
University of California

UNIVERSITY OF CALIFORNIA,
IRVINE

Beyond Similar Code: Leveraging Social Coding Websites

DISSERTATION

submitted in partial satisfaction of the requirements
for the degree of

DOCTOR OF PHILOSOPHY

in Software Engineering

by

Di Yang

Dissertation Committee:
Professor Cristina V. Lopes, Chair
Professor James A. Jones
Professor Sam Malek

2019

© 2019 Di Yang

DEDICATION

To my loving husband, Zikang.

TABLE OF CONTENTS

Page

LIST OF FIGURES **vi**

LIST OF TABLES **viii**

ACKNOWLEDGMENTS **ix**

CURRICULUM VITAE **x**

ABSTRACT OF THE DISSERTATION **xiv**

1 Introduction **1**
- 1.1 Overview . . . 1
  - 1.1.1 Background . . . 1
  - 1.1.2 Research Questions . . . 2
- 1.2 Thesis Statement . . . 3
- 1.3 Methodology and Key Findings . . . 3

2 Usability Analysis of Stack Overflow Code Snippets **6**
- 2.1 Introduction . . . 6
- 2.2 Related Work . . . 9
- 2.3 Usability Rates . . . 11
  - 2.3.1 Snippets Processing . . . 12
  - 2.3.2 Findings . . . 15
  - 2.3.3 Error Messages . . . 18
  - 2.3.4 Heuristic Repairs for Java and C# Snippets . . . 19
- 2.4 Qualitative Analysis . . . 24
- 2.5 Google Search Results . . . 27
- 2.6 Conclusion . . . 30

3 File-level Duplication Analysis of GitHub **32**
- 3.1 Introduction . . . 33
- 3.2 Related Work . . . 37
- 3.3 Analysis Pipeline . . . 40
  - 3.3.1 Tokenization . . . 40
  - 3.3.2 Database . . . 41

3.3.3 SourcererCC 42
3.4 Corpus 43
3.5 Quantitative Analysis 45
3.5.1 File-Level Analysis 45
3.5.2 File-Level Analysis Excluding Small Files 47
3.6 Mixed Method Analysis 48
3.6.1 Most Duplicated Non-Trivial Files 49
3.6.2 File Duplication at Different Levels 52
3.7 Conclusions 58

4 Method-level Duplication Analysis between Stack Overflow and GitHub 60
4.1 Introduction 60
4.2 Related Work 63
4.3 Methodology 66
4.3.1 Method Extraction 66
4.3.2 Tokenization 67
4.3.3 Three Levels of Similarity 68
4.4 Dataset 70
4.5 Quantitative Analysis 72
4.5.1 Block-hash Similarity 74
4.5.2 Token-hash Similarity 75
4.5.3 SCC Similarity 76
4.6 Qualitative Analysis 76
4.6.1 Step 1: Duplicated Blocks 77
4.6.2 Step 2: Large Blocks Present in GitHub and SO 86
4.7 Conclusion 88

5 Analysis of Adaptations from Stack Overflow to GitHub 90
5.1 Introduction 91
5.2 Related Work 93
5.3 Dataset 95
5.4 Adaptation Type Analysis 97
5.4.1 Manual Inspection 97
5.4.2 Automated Adaptation Categorization 101
5.4.3 Accuracy of Adaptation Categorization 102
5.5 Empirical Study 103
5.5.1 How many edits are potentially required to adapt a SO example? . . 103
5.5.2 What are common adaptation and variation types? 104
5.6 Conclusion 106

6 Statement-level Recommendation for Related Code 107
6.1 Introduction 108
6.2 The Opportunity for Aroma 113
6.3 Related Work 116
6.4 Algorithm 118

6.4.1 Definitions . . . 120
6.4.2 Featurization . . . 124
6.4.3 Recommendation Algorithm . . . 127
6.5 Evaluation of Aroma's Code Recommendation Capabilities . . . 137
6.5.1 Datasets . . . 137
6.5.2 Recommendation Performance on Partial Code Snippets . . . 138
6.5.3 Recommendation Quality on Full Code Snippets . . . 139
6.5.4 Comparison with Pattern-Oriented Code Completion . . . 142
6.6 Evaluation of Search Recall . . . 142
6.6.1 Comparison with Clone Detectors and Conventional Search Techniques 144
6.7 Conclusion . . . 146

7 Method-level Recommendation for Related Code 148
7.1 Introduction . . . 148
7.2 Related Work . . . 152
7.3 Data Collection Approach . . . 153
7.3.1 Retrieve similar methods . . . 154
7.3.2 Identify Co-occurring Code Fragments . . . 156
7.3.3 Clustering and Ranking . . . 157
7.4 Dataset . . . 158
7.5 Manual Analysis and Categorization . . . 161
7.6 Chrome Extension for Stack Overflow . . . 169
7.7 Comparison with code search engines . . . 171
7.8 Threats to Validity . . . 172
7.9 Conclusion . . . 173

8 Conclusion 174

Bibliography 177

LIST OF FIGURES

		Page
2.1	Example of Stack Overflow question and answers. The search query was "java parse words in string".	7
2.2	Sequence of operations	13
2.3	Parsable and compilable/runnable rates histogram	16
2.4	Examples of C# Snippets.	17
2.5	Examples of Java Snippets.	17
2.6	Examples of JavaScript Snippets.	18
2.7	Examples of Python Snippets.	18
2.8	Most common error messages for C#	19
2.9	Most common error messages for Java	20
2.10	Most common error messages for JavaScript	20
2.11	Most common error messages for Python	20
2.12	Sequence of operations while applying repairs	23
2.13	Example of an incomplete answer in Stack Overflow	25
2.14	Example of Google Search Result	28
3.1	Map of code duplication. The y-axis is the number of commits per project, the x-axis is the number of files in a project. The value of each tile is the percentage of duplicated files for all projects in the tile. Darker means more clones.	34
3.2	Analysis pipeline.	40
3.3	Files per project.	44
3.4	SLOC per file.	45
3.5	File-level duplication for entire dataset and excluding small files.	47
3.6	Distribution of file-hash clones.	54
3.7	Distribution of token-hash clones.	55
3.8	Δ of file sizes in token hash groups.	56
4.1	Pipeline for file analysis.	65
4.2	Python GitHub projects. LOC means Lines Of source Code, and is calculated after removing empty lines.	71
4.3	Blocks per Post.	72
4.4	Per Block distributions of LOC (top) and unique tokens (bottom), in GitHub and Stack Overflow	73

5.1 Comparison between SO examples in the adaptation dataset and the variation dataset . . . 95
5.2 Code size (LOC) and vote scores on the number of AST edits in a SO example 100
5.3 Frequencies of categorized adaptation types in two datasets . . . 103
5.4 In the lifted template, common unchanged code is retained, while adapted regions are abstracted with *hot spots*. . . . 105

6.1 Distribution of similarity scores used to obtain threshold . . . 114
6.2 Proportion of new code similar to existing code . . . 114
6.3 Aroma code recommendation pipeline . . . 119
6.4 The simplified parse tree representation of the code in Listing 6.4. Keyword tokens at the leaves are omitted to avoid clutter. Variable nodes are highlighted in double circles. . . . 122

7.1 An example of recommending relevant code that complements desired functionality . . . 150
7.2 The pipeline of collecting common co-occurring methods . . . 155
7.3 Distribution of number of GitHub clones . . . 161
7.4 Distribution of number of common co-occurring methods . . . 162
7.5 Distribution of average LOC of common co-occurring methods . . . 162
7.6 Screenshot of Chrome extension . . . 170

LIST OF TABLES

Page

2.1 Operations performed for each language. 13
2.2 Summary of results 16
2.3 Summary of results for C# and Java after single-word snippets removal . . . 17
2.4 Summary of results for C# and Java snippets after repairs 24
2.5 Features used to assess the quality of the snippets. 26
2.6 Usability Rates of Top Results from Google 28

3.1 File-hash duplication in subsets. 35
3.2 GitHub Corpus. 43
3.3 File-Level Duplication. 46
3.4 File-level duplication excluding small files. 48
3.5 Number of tokens per file within certain percentiles of the distribution of file size. 50

4.1 Github Dataset 70
4.2 Stack Overflow Dataset 72
4.3 Block-hash similarity 74
4.4 Token-hash similarity 75
4.5 SCC Similarity 76

5.1 Common adaptation types, categorization, and implementation 98

6.1 Aroma code recommendation examples 112
6.2 Features for selected tokens in Figure 6.4 127
6.3 Categories of Aroma code recommendations 141
6.4 Comparison of recall between a clone detector, conventional search techniques, and Aroma 144

7.1 Complementary method examples 163
7.2 Supplementary method examples 164
7.3 Different implementation example 165
7.4 Categorization of related methods 166
7.5 Related code which can be retrieved by FaCoY 171

ACKNOWLEDGMENTS

I would like to express my deepest appreciation to my advisor, Professor Cristina Lopes. I could not have imagined having a better advisor for my PhD. Crista strongly supported me in every tough time through my PhD, she has more faith and belief in me than I do in myself. Her encouragement is essential to my personal and professional development. With her hands-on guidance in research and writing, I was trained to be a critical thinker and independent researcher. Her great passion in software engineering research inspires me to pursue my own career and make my contributions in this area.

I am very grateful to my dissertation committee members, Prof. James Jones and Prof. Sam Malek, and also my advancement committee members, Prof. Harry Xu and Prof. Ian Harris, for their insightful reviews and valuable advice.

Many thanks to my lab mates, Vaibhav Saini, Pedro Martins, Farima Farahani Farmahini, Hitesh Sajnani, Rohan Achar, and Wen Shen, for taking the time to hear my ideas and brainstorm with me. Thanks the whole Mondego lab for being my sweet harbor.

I would like to extend my sincere thanks to my collaborators, Tianyi Zhang and Prof Miryung Kim from UCLA, Sifei Luan and Satish Chandra from Facebook, and Prof Koushik Sen from UC Berkeley. Tianyi and Miryung helped me to broaden the horizon of my research. I'm very grateful to have such reliable collaborators in the middle of my PhD and our collaboration continues since then. I had an internship at Facebook in Summer 2018. Thanks to Sifei, Satish and Koushik for the mentoring.

I am especially grateful for the love, support and encouragement from my family and friends. Without the unconditional support from my parents, the completion of this dissertation would not be possible. I would like to thank my husband, Zikang, for his love, patience, and continuous support over all the years ever since we've been together. Thanks to Wei Wang and Boyang Wei for being my dear friends and keeping me company during tough times.

I am fortunate to be funded by DARPA MUSE program for more than four years of my research. The majority of the work for this dissertation was done under this support.

CURRICULUM VITAE

Di Yang

Education

Ph.D. Software Engineering *Sep 2013 to Dec 2019*
University of California, Irvine, CA
Dissertation Title: Leveraging Social Coding Websites Towards Searching Beyond Similar Code

M.S. Software Engineering *Sep 2013 to Dec 2018*
University of California, Irvine, CA

B.S. Computer Science *Sep 2009 to Jul 2013*
Beijing Language and Culture University, Beijing, China
Exchange Program Computer Science *Sep 2011 to Jul 2012*
University of Birmingham, Birmingham, UK

Career History

Teaching/Research Assistant, UC Irvine *Sep 2013 - Dec 2019*
Advisor: Cristina Lopes

Software Engineering Intern, Facebook *Summer 2018*
Mentor: Sifei Luan, Koushik Sen, Satish Chandra

Research Intern, SRI International *Summer 2017*
Mentor: Huascar Sanchez, Susmit Jha, Natarajan Shankar

Database Intern, Teradata *Summer 2016*
Mentor: Xin Tang, Renu Varshneya

Research Intern, GrammaTech *Summer 2015*
Mentor: Drew Dehaas, David Melski

Awards and Honors

Distinguished Paper Award: ACM on Programming Languages, Volume 3(OOPSLA), Oct. 2019.

Distinguished Artifact Award: 41st International Conference on Software Engineering (ICSE), May. 2019

Distinguished Artifact Award:ACM Programming Languages, Volume 1(OOPSLA), Oct. 2017.

ACM SIGSOFT CAPS grant: May 2019

UCI AGS travel grant: May 2019

GHC scholarship: Oct 2015

Chair's Award, UCI Informatics: Sep 2013

Publications

- S. Luan, **D. Yang**, C. Barnaby, K. Sen and S. Chandra. Aroma: Code Recommendation via Structural Code Search. In Proceedings of the ACM on Programming Languages, Volume 3(OOPSLA), Oct 2019.
- T. Zhang, **D. Yang**, C. V. Lopes, M. Kim. Analyzing and Supporting Adaptation of Online Code Examples. In proceedings of the 41st International Conference on Software Engineering (ICSE), May 2019.

- V. Saini, F. Farmahinifarahani, Y. Lu, **D. Yang**, P. Martins, H. Sajnani, P. Baldi, and C. V. Lopes. Towards Automating Precision Studies of Clone Detectors. In Proceedings of the the 41st International Conference on Software Engineering (ICSE), May. 2019.
- F. Farmahinifarahani, V. Saini, **D. Yang**, H Sajnani, C. V. Lopes. On Precision of Code Clone Detection Tools. In proceedings of the 26th International Conference on Software Analysis, Evolution and Reengineering (SANER), Feb 2019.
- C. V. Lopes, P. Maj, P. Martins, V. Saini, **D. Yang**, J. Zitny, H. Sajnani, and J. Vitek. Déjàvu: A Map of Code Duplicates on Github. In proceedings of ACM Programming Languages, Volume 1(OOPSLA), Oct. 2017.
- **D. Yang**, P. Martins, V. Saini, and C.V. Lopes. Stack Overflow in Github: Any Snippets There? In proceedings of the 14th International Conference on Mining Software Repositories (MSR), May 2017.
- **D. Yang**, A. Hussain, C. V. Lopes. From Query to Usable Code: An Analysis of Stack Overflow Code Snippets. In proceedings of the 13th International Conference on Mining Software Repositories (MSR), May 2016.
- A. D'Souza, **D. Yang**, C. V. Lopes. Collective Intelligence for Smarter API Recommendations in Python. In proceedings of the 16th International Working Conference on Source Code Analysis and Manipulation (SCAM), Oct 2016.

Public Research Talks & Presentations

- **Presenter**, Towards Automating Precision Studies of Clone Detectors. In Proceedings of the the 41st International Conference on Software Engineering (ICSE), May. 2019.
- **Presenter**, On Precision of Code Clone Detection Tools. In proceedings of the 26th International Conference on Software Analysis, Evolution and Reengineering (SANER), Feb 2019.
- **Presenter**, Stack Overflow in Github: Any Snippets There? In proceedings of the 14th International Conference on Mining Software Repositories (MSR), May 2017.
- **Presenter**, From Query to Usable Code: An Analysis of Stack Overflow Code Snippets. In proceedings of the 13th International Conference on Mining Software Repositories (MSR), May 2016.

ABSTRACT OF THE DISSERTATION

Beyond Similar Code: Leveraging Social Coding Websites

By

Di Yang

Doctor of Philosophy in Software Engineering

University of California, Irvine, 2019

Professor Cristina V. Lopes, Chair

Programmers often write code with similarity to existing code written somewhere. Code search tools can help developers find similar solutions and identify possible improvements. For code search tools, good search results rely on valid data collection. Social coding websites, such as Question & Answer forum Stack Overflow (SO) and project repository GitHub, are popular destinations when programmers look for how to achieve certain programming tasks. Over the years, SO and GitHub have accumulated an enormous knowledge base of, and around, code. Since these software artifacts are publicly available, it is possible to leverage them in code search tools. This dissertation explores the opportunities of leveraging software artifacts from the social coding websites in searching for not just similar, but related, code.

Programmers query SO and GitHub extensively to search for suitable code for reuse, however, not much is known about the usability or quality of the available code from each website. This dissertation first investigates under what circumstances the software artifacts found in social coding websites can be leveraged for purposes other than their immediate use by developers. It points out a number of problems that need to be addressed before those artifacts can be leveraged for code search and development tools. Specifically, triviality, fragility, and duplication, dominate these artifacts. However, when these problems are addressed, there is still a considerable amount of good quality artifacts that can be leveraged.

SO and GitHub are not only two separate data resources, moreover, they together, belong to a larger system of software development process: the same users that rely on facilities of GitHub often seeks support on SO for their problems, and return to GitHub to apply the knowledge acquired. This dissertation further studies the crossover of software artifacts between SO and GitHub, and categorizes the adaptations from a SO code snippet to its GitHub counterparts.

Existing search tools only recommend other code locations that are syntactically or semantically similar to the given code but do not reason about other kinds of relevant code that a developer should also pay attention to, e.g., auxiliary code to accomplish a complete task. With the good quality software artifacts and crossover between the two systems available, this dissertation presents two approaches that leverage these artifacts in searching for related code. AROMA indexes GitHub projects, takes a partial code snippet as input, searches the corpus for methods containing the partial code snippet, and clusters and intersects the results of the search to recommend. AROMA is evaluated on randomly selected queries created from the GitHub corpus, as well as queries derived from SO code snippets. It recommends related code for error checking and handling, objects configuring, etc. Furthermore, a user study is conducted where industrial developers are asked to complete programming tasks using AROMA and provide feedback. The results indicate that AROMA is capable of retrieving and recommending relevant code snippets efficiently. CODEAID reuses the crossover between SO and GitHub and recommends related code outside of a method body. For each SO snippet as a query, CODEAID retrieves the co-occurring code fragments for its GitHub counterparts and clusters them to recommend common ones. 74% of the common co-occurring code fragments represent related functionality that should be included in code search results. Three major types of relevancy–*complementary*, *supplementary*, and *alternative* methods, are identified.

Chapter 1

Introduction

1.1 Overview

1.1.1 Background

When programmers look for how to achieve certain programming tasks, social coding websites, such as Question and Answer forum Stack Overflow (SO) and project repository GitHub, are popular destinations. The popularity and relevance of SO and GitHub are well known within the programming community. Both websites are extensively queried when programmers search for suitable code for reuse. Moreover, they, together, belong to a larger system of software development process: the same users that rely on facilities of GitHub often seeks support on SO for their problems, and return to GitHub to apply the knowledge acquired.

Studies have been done on the immediate use by developers of the two websites: GitHub as a project management repository and SO as a Q&A forum. For example, Kalliamvakou et.al analyzed pull requests, commits and issues on GitHub [65, 132] Bajaj et. al mined

related questions in SO [21], AutoComment [147] automatically generates comments, EnTagRec [144] is an automatic tag recommender. Other studies focus on people, or social components for the two websites. Kalliamvakou et. al studied the impact of networking behaviors on collaborations in GitHub [65], Thung et.al graphed developer-developer and project-project relations and identified influential developers and projects [128], Vasilescu et.al analyzed gender and tenure diversity on GitHub [138], how SO changes knowledge sharing [139], and the relationship between SO askers and GitHub committers [137].

Over the years, the social coding websites SO and GitHub have accumulated an enormous knowledge base of, and around, code. Since these software artifacts are publicly available, it is possible to leverage them in code search tools. Recent research have already paid special attention to the code. Ray et.al studied the influence of language typing on quality of software quality in GitHub [106]. Ponzenelli et.al intergrated SO into IDE [102]. An et.al pointed out the violation of license when copy&pasting SO snippets [18]. Several work have found SO snippets are incomplete and inadequate [123, 152, 156]. These studies targeted at integrating SO snippets into client systems, however, not much is known about the usability of the available code from each website, the impact that these two systems have on each other, and how we can leverage them in code search and recommendation tools.

1.1.2 Research Questions

Before reusing these artifacts in code search, we need to know the quality of these artifacts. In this dissertation, I investigate under what circumstances the software artifacts found in social coding websites can be leveraged for purposes other than their immediate use by developers. There is an increasing number of research and tools that rely on software artifacts from SO and GitHub. Without understanding the artifacts themselves beforehand, those studies and tools may end up with skewed analysis results or disappointing tool performance.

To be specific, I have been investigating the following questions:

1. What is the quality of these software artifacts? Do they have problems that need to be known before they can be used for code search tools?
2. Is there crossover of software artifacts between SO an GitHub? In what form?
3. How can we leverage the artifacts from SO and GitHub and the crossover between SO and GitHub in code search and recommendation tools? Can we search more than just similar code from these artifacts?

1.2 Thesis Statement

Software artifacts from social coding websites are publicly available to be leveraged in code search tools, however, a number of problems need to be addressed before those artifacts being leveraged. Specifically, triviality, fragility, and duplication, dominate these artifacts. When these problems are addressed, there is still a considerable amount of good quality artifacts. Moreover, there is crossover between different social coding websites which shows the adaptation behaviors in real-life programming. The good quality artifacts from different websites, together with the crossover, can be leveraged in searching not just similar, but related, code.

1.3 Methodology and Key Findings

In this dissertation, I start with studying the usability of SO snippets and duplication within SO and within GitHub. Then I find similar code pairs between SO and GitHub to analyze their relationship in terms of crossover of code. I carefully investigate the adaptation types

between the SO snippets and their GitHub similar counterparts. Finally I propose two code recommendation tools as examples of leveraging the artifacts from SO and GitHub and the crossover between SO and GitHub.

In Chapter 2, I performed the usability analysis of SO snippets. Usability is defined based on the standard steps of parsing, compiling and running source code to show the effort that would (potentially) be required to leverage the snippets as-is. I studied the percentages of parsable, compilable and runnable snippets for each of the four most popular programming languages (C#, Java, JavaScript, and Python). There is a significant difference in usability between the statically-typed languages and the dynamically-typed languages, the latter being far less usable.

In Chapter 3, for Java and Python, I performed file-level and project-level duplication analysis on GitHub repositories. I provided three levels of similarity: a hash on the block, a hash on tokens, and an 80% token similarity. These capture the case where entire blocks are copied as-is, smaller changes are made in spacing or comments, and more meaningful edits are applied to the code.

The studies show that fragile code needs to be repaired in SO before being leveraged in code search tools; trivial code in SO and files in GitHub should be removed; duplication dominates GitHub files and de-duplication needs to be performed. Addressing these problems avoids noise and bias in the datasets and provide good quality software artifacts for code search tools.

In Chapter 4 and 5, I studied the method-level similar code between SO and GitHub for Python and Java. Again, three levels of similarity are provided. For Java, I further conducted a detailed manual inspection on the adaptations from SO to GitHub. I built an automated adaptation analysis technique to categorize syntactic program differences into different adaptation types. This study serves as a proof that there is meaningful crossover

between SO and GitHub in terms of software artifacts, which can be leveraged for research and development.

In Chapter 6, I proposed a code search and recommendation tool, Aroma, which leverages good quality software artifacts from SO and GitHub. Aroma indexes a huge code corpus including thousands of open-source projects from GitHub, takes a partial code snippet as input, searches the corpus for method bodies containing the partial code snippet, and clusters and intersects the results of the search to recommend a small set of succinct code snippets which both contain the query snippet and appear as part of several methods in the corpus. Aroma is evaluated on 2000 randomly selected queries created from the corpus, as well as 64 queries derived from code snippets obtained from SO.

In Chapter 7, I proposed one possibility of leveraging the crossover between SO and GitHub in recommending related code. I first constructed a large data set of 21K groups of similar code written in Java, using SO code snippets as queries and identifying their counterparts in GitHub via clone detection. For more than half of these SO code snippets, their GitHub counterparts share common code that co-occurs in the same Java file yet not similar to the original queries. I manually inspected a random sample of 50 commonly co-occurring code fragments and found 74% of them represent relevant functionality that should be included in code search results. Furthermore, I identified three major types of relevant co-occurring code—complementary, supplementary, and alternative functions. This study result calls for a new search engine that accounts for such code relevance beyond code similarity.

Finally, Chapter 8 concludes the work and contributions for this dissertation.

Chapter 2

Usability Analysis of Stack Overflow Code Snippets

The material in this chapter is from the following paper, and is included here with permission from ACM.

D. Yang, A. Hussain, C. V. Lopes. From Query to Usable Code: An Analysis of Stack Overflow Code Snippets. In proceedings of the 13th International Conference on Mining Software Repositories (MSR), May 2016.

2.1 Introduction

Research shows that programmers use web searches extensively to look for suitable pieces of code for reuse, which they adapt to their needs [52, 100, 135]. Among the many good sites for this purpose, Stack Overflow (SO, from here onwards) is one of the most popular destinations in Google search results. Over the years, SO has accumulated an impressive amount of programming knowledge consisting of snippets of code together with relevant

Converting a sentence string to a string array of words in Java

8

3

I need my Java program to take a string like:

```
"This is a sample sentence."
```

and turn it into a string array like:

```
{"this","is","a","sample","sentence"}
```

No periods, or punctuation (preferably). By the way, the string input is always one sentence.

Is there an easy way to do this that I'm not seeing? Or do we really have to search for spaces a lot and create new strings from the areas between the spaces (which are words)?

java string spaces words

share | improve this question

asked Jan 12 '11 at 22:44

372 • 1 • 9 • 19

You may also want to look at the guava Splitter class: guava-libraries.googlecode.com/svn/trunk/javadoc/com/google/... – dkarp Jan 12 '11 at 22:51

add comment

8 Answers

active oldest votes

15

String.split() will do most of what you want. You may then need to loop over the words to pull out any punctuation.

For example:

```
String s = "This is a sample sentence.";
String[] words = s.split("\\s+");
for (int i = 0; i < words.length; i++) {
    // You may want to check for a non-word character before blindly
    // performing a replacement
    // It may also be necessary to adjust the character class
    words[i] = words[i].replaceAll("[^\w]", "");
}
```

share | improve this answer

answered Jan 12 '11 at 22:47

19.7k • 3 • 48 • 77

Figure 2.1: Example of Stack Overflow question and answers. The search query was "java parse words in string".

natural language explanations. Besides being useful for developers, SO can potentially be used as a knowledge base for tools that automatically combine snippets of code in order to obtain more complex behavior. Moreover those more complex snippets could be retrieved by matches on the natural language (i.e. non-coding information) that enriches the small snippets in SO.

As an illustrative example, consider searching for "java parse words in string" using Google Web search. This yields several results in SO, one of which is shown in Figure 2.1. The snippet of code provided with the answer that was accepted is almost executable as-is by copy-paste. The job of programmers becomes, to a large extent, to glue together these snippets of code that do some generic functionality by themselves. The fact that a Web search engine returned this SO page as one of the top hits for our query closes in on one of the hardest parts of program synthesis, namely the expression of complex program specifications. Hence, it is conceivable that tools might be developed that would do that gluing job automatically

given a high-level specification in natural language.

In pursuing this goal, the first challenge one faces is whether, and to what extent, the existing snippets of code that are suggested by Web search results are *usable* as-is. If there are not enough usable snippets of code, the process of repairing them automatically for further composition may be out of reach. This paper presents research in this direction, by showing the results of our investigation of the following questions:

(1) How usable are the SO code snippets?

(2) When using Web search engines for matching on the natural language questions and answers around the snippets, what percentage of the top results contain usable code snippets?

In order to compare the *usability* of different pieces of code, we need to define what *usability* is in the first place. We classified snippets of code based on the effort that would (potentially) be required by a program generation tool to use the snippet as-is. Usability is therefore defined based on the standard steps of parsing, compiling and running source code. For each of these steps, if the source code passes, the more likely it is that the tool can use it with minimum effort.

Given this definition of usability, there are situations where a snippet that does not parse is more useful than the one that runs, but passing these steps assures us of important characteristics of the snippet, such as the code being syntactically and structurally correct, or all the dependencies being readily available, which are of surmount importance for automation.

We first study the percentages of parsable, compilable and runnable (where these steps apply) snippets for each of the four most popular programming languages (C#, Java, JavaScript, and Python).[1] From the results, we saw a significant difference in repair effort (usability)

[1]Based on the RedMonk programming language popularity rankings as of January 2015, four of the most popular programming languages are Java, C#, JavaScript and Python. We choose these four, also as representatives of statically-typed (the first two) and dynamically-typed languages (the last two).

between the statically-typed the dynamically-typed languages, the latter being far less effort. Next, we focused on the best performing language (Python) and conducted a 3-step qualitative analysis to see if the runnable snippets can actually answer questions correctly and completely. Finally, in order to close the circle, we use Google Search in order to find out the extent to which the SO snippets suggested by the top Google results are usable. Being able to find a large percentage of usable snippets among the top search results for informal queries, the idea of automating snippet repair and composition, and finding those synthetic pieces of code via informal Web queries becomes within the realm of possibility.

The remainder of this paper is organized as follows. In Section 2.3, we present the overall research methodology and environment of our work. The results of qualitative analysis are explained in Section 2.4. In Section 2.5 we investigate the usability and quality of top results from Google Web search. In Section 2.2, we present the related work in the areas of SO analyses, enhancing coding environments, and automated code generation. Section 2.6 concludes the paper.

2.2 Related Work

Various studies have been done on SO, but focus primarily on user behavior and their interactions with one another. These works made attempts at identifying correlations between different traits of SO users. For example [86] showed a correlation between the age and reputation of a user by exploring hypotheses such as the fact that older users having a bigger knowledge of more technologies and services. Shaowei et al. [143] provided an empirical study on the interactions of developers in SO, revealing statistics on developers' questioning and answering habits. For instance, they found that a few developers ask and answer many questions. This social research might be important for our prioritization of snippets of code.

Among the works that utilize code available in the public domain for enhancing development is that of Wong et al. [147]. They devised a tool that automatically generates comments for software projects by searching for accompanying comments to SO code that are similar to the project code. They did so by relying on clone detection, but never tried to actually use the snippets of code. This work is very similar to Ponzanelli et al. [102] in terms of the approach adopted. Both mine for SO code snippets that are clones to a snippet in the client system, but Ponzanelli et al.'s goal was to integrate SO into an Integrated Development Environment (IDE) and seamlessly obtain code prompts from SO when coding.

Ponzanelli was involved in another work [20], where they presented an Eclipse plugin, Seahawk, that also integrates SO within the IDE. It can add support to code by linking files to SO discussions, and can also generate comments to IDE code. Similarly, Suresh et al. [125] present a tool called PARSEWeb that assists in reusing open source frameworks or libraries by providing an efficient means for retrieving them from open source code.

With regards to assessing the usability of code, our central motivation, our study comes close to Nasehi et al.'s work [89]. They also analyzed SO code with the motivation of finding out how easy it is to reuse it. In particular, they delved into finding the characteristics of a good example. The difference for our approach was their criteria for assessing the usability of the code. They adopted a holistic approach and analyzed the characteristics of high voted answers and low voted answers. They enlisted traits related to a wide range of attributes of the answers by analyzing both the code and the contextual information. They looked into the overall organization of the answer - the number of code blocks used in the answer, the conciseness of the code, the presence of links to other resources, the presence of alternate solutions, code comments, etc. The execution behavior of the code was not among their usability criteria.

Semi-automatic or automatic programming, a development realm towards which this work takes an initial step, has also been explored in different ways by software practitioners. For

instance, Budinskey et al. [31] and Frederick et al. [50], analyze how design patterns could assist in automatically generating software code. Other similar works include [95], [154], and [96]. The similarity in these works is that structured abstractions of code provide a good indicator about the actual implementation. They built tools that exploit this narrative and generate implementations from design patterns. Such a strategy would be highly challenging to use when trying to reproduce usable SO code (with respect to compilation), as those codes are usually small snippets that do not follow familiar patterns.

2.3 Usability Rates

This section describes our usability study of SO snippets. We elaborate our goal, describe the characteristics of the extracted snippets, and present the operations that were carried out on the snippets of each language. We also describe the libraries that were used to process the snippets for each of the languages, and highlight the limitations found[2]. In 2.3.2 and 2.3.3, we present the usability rates and error messages for each language.

Our goal is to compare the usability rates for the snippets of four programming languages (C#, Java, JavaScript, and Python) as they exist in SO, i.e., in small snippets of code. We also want compare the languages regarding their static or dynamic nature, and target the most usable language. In our study, snippets in Java and C#, which are statically-typed, are parsed and compiled in order to assess their usability level. JavaScript does not have the process of compilation, so we investigate only the parsing and running levels for it. Python is also a dynamic language but it can be compiled. However, this step is not as important as it is for Java and C#, as important errors (such as name bindings) are not checked at compile time. As such, for Python, like for JavaScript, we assess usability only by parsing and running the snippets.

[2]Note to reviewers: code and data for this study are available upon request, and will be made publicly available upon publication of this paper.

2.3.1 Snippets Processing

All snippets were extracted from the dump available at the Stack Exchange data dump site[3].

In SO both questions and answers are considered as *posts*, which are stored with unique ids in a table called `Posts`. In this table, posts that represent questions and answers are distinguished by the field `PostTypeId`. Information about posts can be obtained by accessing the field `Body`.

There are two types of answers in SO, *accepted answer* and *best answer*. An accepted answer is the answer chosen by the original poster of the question to be the most suitable. All question posts have an `AcceptedAnswerId` field from which we can identify accepted answers when these exist. The best answer is the one which has the most number of votes from other SO users. The vote count is stored in the field `ViewCount` of the table `Posts`. Thus, an accepted answer may not always be the best answer.

Finally, in SO, questions are tagged with their subject areas, which include relevant information such as the language or the domain where the question is relevant (networking, text processing, etc.). We get the language information for each accepted answer from these tags, one example on how the social nature of SO helps categorizing and selecting pieces of code.

In this work, we only include snippets found in all **accepted answers**. We choose accepted these as we value the agreement from the original poster, accepting the fact that it is very likely this answer resolved the original problem. For all posts for a language we were interested in, we used the the markdown `<code>` to extract the code snippets from the field `Body`.

In Table 2.1 we present the operations we performed to analyze and rate each of language. All the snippets from all languages were parsed, but depending on the static or dynamic

[3]https://archive.org/details/stackexchange, obtained on April 2014.

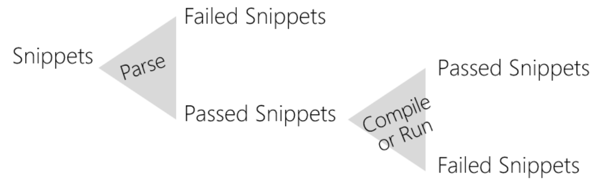


Figure 2.2: Sequence of operations

nature of the language we either compiled it and analyzed the (possible) errors, or ran the language (below we detail these processes).

Table 2.1: Operations performed for each language.

Operation	C#	Java	JavaScript	Python
Parse	x	x	x	x
Compile	x	x		
Run			x	x

Figure 2.2 shows the order in which these operations are performed. We compile (or run) only those snippets which passed parsing, since snippets which are unparsable have syntactic errors and therefore are also non-compilable/non-runnable.

We used a set of tools and APIs to process the snippets in the various languages, which we present next:

2.3.1.1 C#

For parsing we utilize a tool called Roslyn by Microsoft to obtain the parsable snippets. Roslyn provides for the Visual Studio languages rich APIs for different phases of compilation. In particular, Roslyn provides us with the API for getting the abstract syntax tree, which is the landmark of parsing process. Syntax errors will be detected in this step.

Compiling C# programmatically can be easily done by using a functionality provided by the .NET Framework and found in the `Microsoft.CSharp` and `System.CodeDom.Compiler`

namespaces. We need to call the function that compiles the code, and results of whether a snippet compiles or not are returned together with errors if applicable.

2.3.1.2 Java

Eclipse's JDT (Java Development Tools) Parser (ASTParser) and the Javax.Tools were used for the parsing and compiling processes, respectively. We use JDT1.7 in our experiments since it's the latest version for our data dump.

Using the JDT's `ASTParser` we generated abstract syntax trees of the snippets, and any parse errors found during the process were extracted via the `IProblems` interface.

Javax.Tools compilation functionality first creates a dynamic source code file object of the Java snippet, from which it generates a list of compilation units, which are passed as parameters to the `CompilationTask` object for compilation. Issues during compilation are stored in a `Diagnostics` object. Issues could be of the following kinds: `ERROR, MANDATORY_WARNING, NOTE, OTHER, WARNING`. We only look for issues which are of kind `ERROR`, as they are the ones more likely prevent the normal completion of compilation.

2.3.1.3 JavaScript

A reflection of the SpiderMonkey parser is included in the SpiderMonkey JavaScript Shell and is made available as a JavaScript API. It parses a string as a JavaScript program and returns a `Program` object representing the parsed abstract syntax tree. Syntax errors are thrown if parsing fails. JavaScript Shell has also a built-in function `eval()` to execute JavaScript code, which we used if parsing succeed.

A limitation of the SpiderMonkey parser is that it terminates the processing of a snippet right when it encounters the first error. Therefore, it does not identify *all* the errors in a

snippet, only the first one, but this suffices to detect problems in the code.

2.3.1.4 Python

Python's built-in AST module and `compile()` method can help us parse code strings. Python is a special language among dynamic languages: it has the process of building abstract syntax tree into Python code objects, so it has the `compile` function. But when we specify one of the function parameters to be AST only, it only parse the code by building the AST. `exec` statement provides functionality to run code strings.

One problem we encountered in processing Python snippets was that Python2 and Python3 have some incompatible language features. To deal with snippets written in different versions of Python, and to avoid being biased when rating these pieces of code, we first examined all Python snippets under Python2 engine, and examined the unparsable ones again under the Python3 engine and combine the results. The Python libraries share the same limitation as JavaScript's SpiderMonkey; they do not catch *all* the errors in a snippet, only the first one.

2.3.2 Findings

We present the results that were obtained after the initial parsing and compiling (or running) of the snippets.

Table 2.2 and Figure 2.3 shows the summary of usability results of all the snippets. A total of 3M code snippets were analyzed. Python and JavaScript proved to be the languages for which the most code snippets are usable: 537,767 (65.88%) JavaScript snippets are parsable and 163,247 (20.00%) of them are runnable; for Python, 402,249 (76.22%) are parsable and 135,147 (25.61%) are runnable. Conversely, Java and C# proved to be the languages with the lowest usability rate: 129,727 (16.00%) C# snippets are parsable but only 986 (0.12%) of

Table 2.2: Summary of results

	C#	Java	JavaScript	Python
Total Snippets	810,829	914,974	816,227	527,774
Parsable Snippets	129,727 (16.00%)	35,619 (3.89%)	537,767 (65.88%)	402,249 (76.22%)
Compilable Snippets	986 (0.12%)	9,177(1.00%)	–	–
Runnable Snippets	–	–	163,247 (20.00%)	135,147 (25.61%)

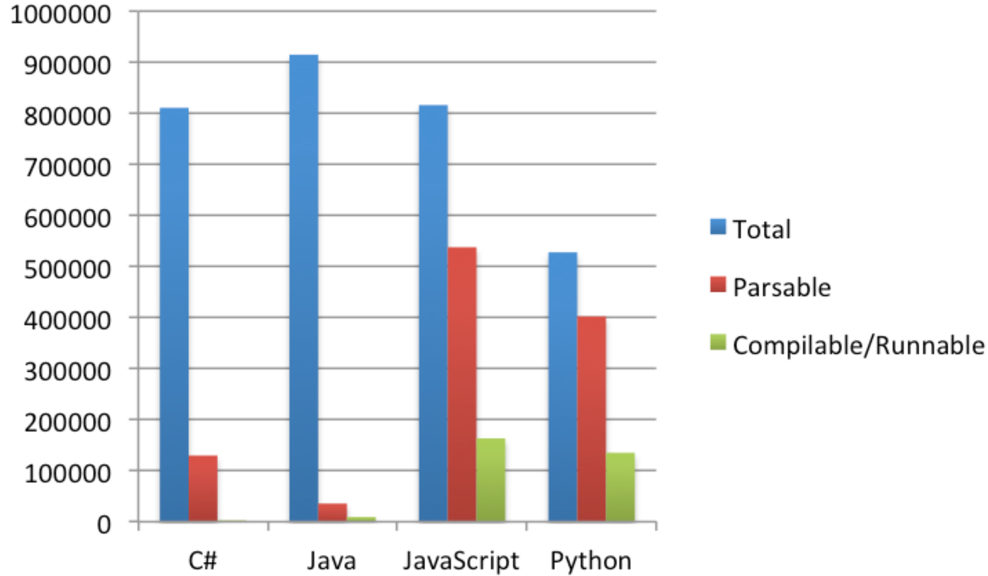


Figure 2.3: Parsable and compilable/runnable rates histogram

them are compilable; for Java, only 35,619 (3.89%) are parsable and 9,177 (1.00%) compile.

As a result of finding such low parsable and compilable rates for Java and C#, we removed Java and C# snippets that only contained single words (i.e. tokens without non-alphabetical characters). The rationale behind this step was to that a single word in C# or Java is too insignificant a candidate for composability; by ignoring those snippets we might improve the usability rates for these two languages. We then parsed and compiled the remaining snippets, the results of which are shown in Table 2.3. We see that the rates of usability improve for both languages, and for both parsing and compilation. For Java, the parsable rate increases from 3.89% to 6.22%, and the compilable rate increases from 1.00% to 1.60%. For C#, the parsable rate increases from 16.00% to 25.18%, and the compilable rate increases from 0.12% to 0.19%.

Table 2.3: Summary of results for C# and Java after single-word snippets removal

	C#	**Java**
Total snippets after removal	514,992	572,742
Parsable	129,691 (25.18%)	35,619 (6.22%)
Compilable	986 (0.19%)	9,177 (1.60%)

```
C#
Parsable, Compilable
using System;
namespace ConsoleApplication1
{
    class Program
    {
        static void Main(string[] args)
        {
            MyClass foo = new MyClass();
            Console.ReadLine();
        }
    }
    class BaseClass
    {
        public BaseClass()
        {
            Console.WriteLine("BaseClass constructor called.");
        }
    }
    class MyClass : BaseClass
    {
        public MyClass()
        {
            Console.WriteLine("MyClass constructor called.");
        }
    }
}
Parsable, Uncompilable
public void RaisePostBackEvent(string eventArgument) { }

Error(s):
Expected class, delegate, enum, interface, or struct
Unparsable
Console.ReadLine()

Error(s):
Type or namespace definition, or end-of-file expected
A namespace cannot directly contain members such as fields or methods
```

Figure 2.4: Examples of C# Snippets.

```
Java
Parsable, Compilable
public class Gallow extends JPanel {
  public paintComponent(Graphics g){
    g.fillOval(10, 20, 40, 25);
  }
}
Parsable, Uncompilable
public enum Command
{
  START(StartCommand.class),
  END(EndCommand.class);

  private Class<? extends CommandInterface> mappedClass;
  private Command(Class<? CommandInterface> c) { mappedClass = c;
  public CommandInterface getInstance()
  {
    return mappedClass.newInstance();
  }
}

Example Error:
cannot find symbol [symbol: class CommandInterface,
location: class Command]
Unparsable
List<?> list = (List<?>) object;

Example Error:
Syntax error on token "List", interface expected before this toke
```

Figure 2.5: Examples of Java Snippets.

Figures 2.4, 2.5, 2.6, and 2.7 show examples of SO snippets that passed and failed for each language. For those that failed, we also show the generated error messages. The examples are representative of the most common error messages.

The unparsable Python snippet in Figure 2.7 also illustrates a common occurrence in SO posts, where the example code is given more or less as pseudo-code that mixes the syntax of several languages.

JavaScript

Parsable, Runnable

```
function setColor(element, color){
    element.style.backgroundColor = color;
}
```

Parsable, Non-runnable

```
expression.call

Error(s):
ReferenceError: expression is not defined
```

Unparsable

```
<%= yield :window_name %>

Error(s):
SyntaxError: expected expression, got '<'
```

Figure 2.6: Examples of JavaScript Snippets.

Python

Parsable, Runnable

```
import sys
sys.stdout.write('\a')
sys.stdout.flush()
```

Parsable, Non-runnable

```
result = getattr(foo, 'bar')()

Error(s):
name 'foo' is not defined
```

Unparsable

```
p = Pita()
while p( next() ) != END:
   // do stuff with p.pocket!

Error(s):
invalid syntax
```

Figure 2.7: Examples of Python Snippets.

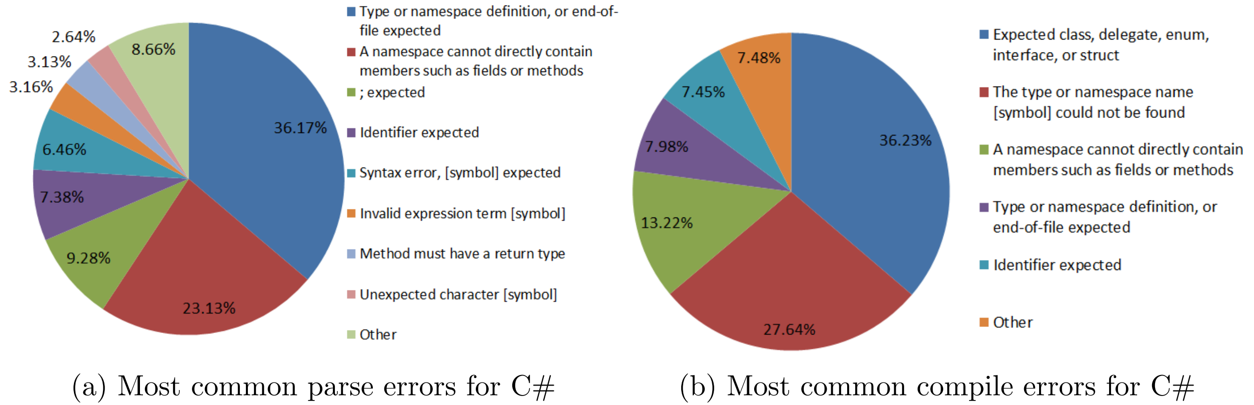
2.3.3 Error Messages

During the usability analysis process, we log the common errors of the four languages. In total, we collected 3,347,674 parse errors and 359,783 compile errors for C#, 1,417,910 parse errors and 199,489 compile errors for Java, 278,460 parse errors and 374,520 runtime errors for JavaScript, and 125,525 parse errors and 267,102 runtime errors for Python.

The common error messages for C# are shown in Figure 2.8a and 2.8b, for Java in Figure 2.9a and 2.9b, for JavaScript in Figure 2.10a and 2.10b, and for Python in Figure 2.11a and 2.11b. The error messages are listed in descending order of percentage. The token '[symbol]' is just a replacement for various specific strings appeared in error messages.

The error messages shown for Java and C# were obtained on the collection of snippets without single-words. It is important to note that the libraries used for Python and JavaScript can generate at most one error message for a snippet, so they do not show all problems that each snippet may have.

Main syntax problems are shown in parsing errors, for all four languages. For example for JavaScript, 50% of the parsing errors are not getting an expression. For Java, 25% of the



(a) Most common parse errors for C# (b) Most common compile errors for C#

Figure 2.8: Most common error messages for C#

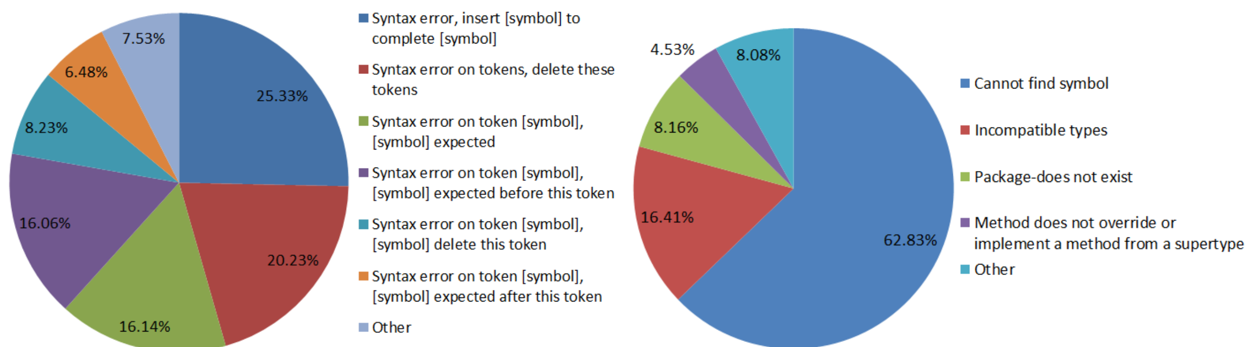
parsing errors are tokens to be inserted.

Errors more related to code context, such as missing symbols, are revealed in compiling or running process. For C#, "type or namespace" is a main issue for usability. For Java, "cannot find symbol" dominates compiling error messages. When running a JavaScript snippet, we are most likely to stop at a reference error, while for Python, the most common runtime error is a specific name not defined.

The purpose for logging the error messages is to provide a knowledge base for repairing codes and increasing usability rates in the future. For example one of the main parsing errors for C# is missing semicolons, then a heuristic repair to C# codes to improve parsable rate can be locating missing semicolons and append them. In next section, we give example of two heuristic repairs for Java and C# snippets.

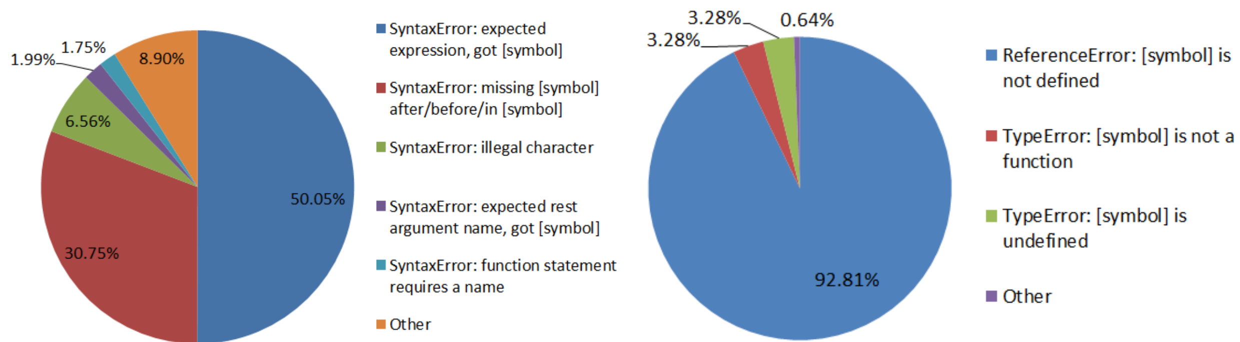
2.3.4 Heuristic Repairs for Java and C# Snippets

From the preliminary results above, we can see that the parsing rates for Python and JavaScript are significantly better than Java and C#. The parsing errors reveal the main syntax problems, while the compiling errors given above are more related to code context, such as missing symbols. In this case, compiling errors are hard to fix, because we need to



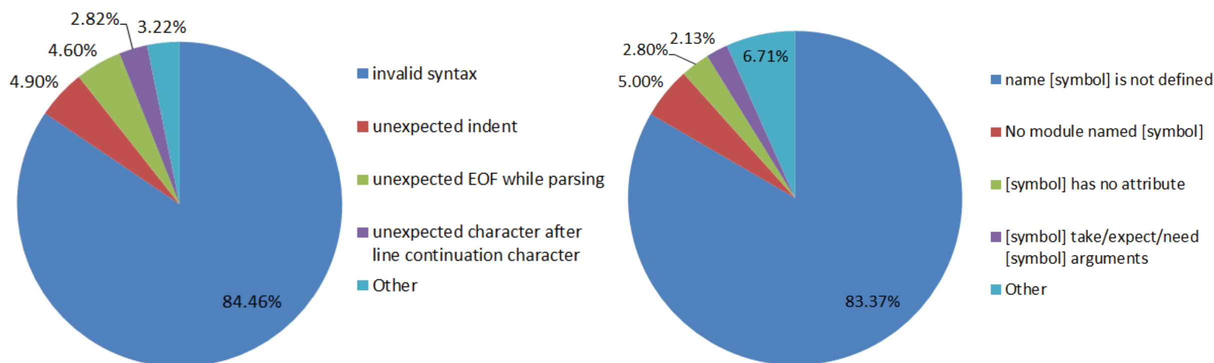
(a) Most common parse errors for Java (b) Most common compile errors for Java

Figure 2.9: Most common error messages for Java



(a) Most common parse errors for JavaScript (b) Most common runtime errors for JavaScript

Figure 2.10: Most common error messages for JavaScript



(a) Most common parse errors for Python (b) Most common runtime errors for Python

Figure 2.11: Most common error messages for Python

look into the specific snippet to complement the missing symbols.

Based on the common error messages for Java and C#, we implemented two heuristic repairs on Java and one repair on C# snippets in order to improve their parsing and compilation rates.

2.3.4.1 Repair 1 - Class

Many Java snippets consist just of Java code without it being properly encapsulated in a class or a method. The `class` construct is essential for Java snippets. The `class` repair fixes Java code snippets that were found to be missing a class construct based on a heuristic check. This heuristic check works as follows: if the code snippet is found to contain any of the tokens `import`, `package`, or `class`, we assume that the `class` construct already exists in the snippet. The rationale behind this heuristic is that, based on our observations of the snippets, tokens `import` and `package` form scaffolding information of code in SO and are not the focus of SO answers. Hence any code that uses one of them is likely to use the `class` construct also. We also assume if a token `class` is present in a code, it exists with enclosing braces and as a keyword and not a part of a string or comment.

Example:

```
\\Repair 1 Candidate
public void main(String args []){
  System.out.println("Hello World");
}
\\After Repair 1
class Program{
  public void main(String args []){
```

```
    System.out.println("Hello World");
  }
}
```

Unlike Java, C# does not require a class construct for error-free parsing and compilation, and thus this repair was only applied to Java snippets.

2.3.4.2 Repair 2 - Semicolon

Java and C# statements require a semicolon (";") at the end in order to parse and compile correctly. To decide whether a ";" should be added to a statement, we run a set of heuristic checks on each line of the snippet; we add the semicolon if all the following conditions are true:

1. If the line does not contain any of the tokens `;`, {, `(`, and
2. if the line does not contain any of the tokens `class`, `if`, `else`, `do`, `while`, `for`, `try`, `catch`, and
3. if the line does not end with the tokens = and }.

With check 1, we avoid double-adding ";" and avoid adding a ";" at the line of an opening brace, before the opening brace has been closed. With check 2 and 3 we avoid corrupting originally parsable code. With check 2 we avoid the following situation:

```
\\Repair 2
try; <--  will corrupt
{  <code>
}catch...
```

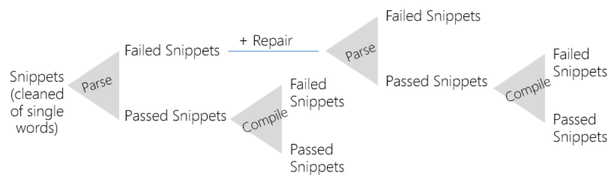


Figure 2.12: Sequence of operations while applying repairs

With check 3 we avoid the following situations:

```
\\Repair 2 inside assignment
 Double s_dev = ; <-- will corrupt
     Math.pow(sum(mean_sq(al))/al.size(),0.5);
\\Repair 2 between if-else
if ()
{   <code>
} ; <-- will corrupt
else
{   <code>
}
```

2.3.4.3 Study Workflow for Repairs

The workflow for studying the effect of repairs is similar to the one used for the initial parsing/compiling processes (as shown in Figure 2.2), except that now we have also incorporated repairs. The process is depicted in Figure 2.12.

Like the workflow of Figure 2.2, here at each stage we only compile snippets that have passed the prior step. Failed snippets are repaired and parsed again. The snippets that succeed at parsing are in turn compiled. For Java, we carry out two repairs sequentially, whereas for C# one repair is applied.

Table 2.4: Summary of results for C# and Java snippets after repairs

	C#	Java
Total snippets after removal	514,992	572,742
Parsable snippets after repairs	135,421 (26.30%)	110,203 (19.24%)
Compilable snippets after repairs	986 (0.19%)	17,286 (3.02%)

2.3.4.4 Results after Repairs

Table 2.4 shows the parsing and compilation results obtained for C# and Java after repairing the snippets. Again, the numbers here reflect the collection of non-single-word snippets.

Although the repairs did not significantly increase the usability rates for C#, the improvements were quite significant for parsing Java snippets. The parse rate of C# improved by only 1.12% (from 25.18% to 26.30%), whereas for Java the improvement was 13.02% (from 6.22% to 19.24%). The compilation rate did not change for C#, whereas for Java it improved by 1.42% (from 1.6% to 3.02%). There's a significant improvement on the parsable rate of Java.

Again, our approaches are heuristic, and may break some previously parsable or compilable snippets. But we can still see an increase in usability rates.

Even though the parsing and compilation rates improved for Java, the number of usable snippets is still one order of magnitude lower than the numbers for JavaScript and Python.

2.4 Qualitative Analysis

Based on the usability and popularity, we choose Python as the target language for further analysis. In order to investigate whether the runnable Python snippets can answer the

Python appending to two lists when it should only append to one

0

I have a list called teams which contains two objects, these are objects of the same class and they both have a "members" list. I am appending to these lists individually. See Fight.AddParticipant but the two participant objects I'm appending seem to end up in both of the team objects, which is unintended behavior. Why is this happening?

Code:

```
class Fight:
    participants = []
    teams = []
    attacked = []
    fighting = 0

    def MakeTeams(self, team):
        self.teams.append(team)

    def NumParticipants(self, teamnum = None):
        if (teamnum != None):
            return len(self.teams[teamnum].members)
        else:
            return len(self.participants)


    def AddParticipant(self, participant, team):
        self.participants.append(participant)
        ref = self.participants[-1]
        self.teams[team].members.append(ref)
        # print self.teams[1].members[0].name
```

3

In all of your classes, you want to initialize instance variables like this:

```
def __init__(self):
    self.participants = []
    self.teams = []
    self.attacked = []
    self.fighting = 0
```

That way, they are separate for each fight, participant, team instead of shared for all fights, participants, or teams.

Figure 2.13: Example of an incomplete answer in Stack Overflow

questions *correctly* and *completely*, we perform a 3-step qualitative analysis on randomly selected snippets. By *correctness*, we mean the snippet giving a concise solution to the question; for specific coding questions with bugs, as in Figure 2.13, the answer is *correct* if it points out the erroneous part and fixes particular lines of code. By *completeness*, we mean that the snippet itself is a full answer to the question; we do not need to add any additional code to answer the question. Figure 2.13 is an example of correct but incomplete answer, the snippet fixes the bug in the original code but we have to mix the question and answer snippets to get the full answer.

The 3-step qualitative analysis is as following:

Step 1

We randomly chose 50 runnable Python snippets. For each snippet, we investigate the features listed in Table 2.5. We found out that the proportion of snippets that answer the question is low (16%). We discovered a strong correlation between single word snippets and

Table 2.5: Features used to assess the quality of the snippets.

1. Votes for the question
2. Votes for accepted answer
3. Total number of answers
4. Is the accepted answer also the best answer?
5. Questioner's reputation score
6. Answerer's reputation score
7. Does the title correctly summarize the question described?
8. Is the question's description clear?
9. Is it a specific coding question?
10. Does the snippet answer the question correctly and completely?
11. Is it a single word snippet?
12. Is it a single line snippet?
13. Is there any surrounding context/explanation?
14. Number of comments
15. Is there any questioner's compliment in comments?
16. Question's tags

snippets answering the question, that is, among the 50 selected snippets, none of single word snippets answer the question, and all of the snippets that answer the question are non-single word snippets. The proportion of single word snippets is 64%.

Step 2

Based on the results of Step 1, we removed single word snippets from all runnable Python snippets, and then randomly chose another 50 snippets. We investigated the same aspects as in Step 1, except for No.11 (Is it a single word snippet?).

After removing single word snippets, the proportion of snippets that answer the question increases to 44%. From Step 2, we discovered another negative correlation between single line snippets and snippets answering the question. Among the 21 snippets that answer the question, 19 are multiple line snippets, and among the 20 single line snippets, only 2 answer

the question.

Step 3

Finally, we removed the single line snippets, and chose 50 snippets randomly again. We investigated the same aspects as in Step 1, except for No.11 (Is it a single word snippet?) and No.12 (Is it a single line snippet). Again, the proportion of snippets that answer the question increases, to 66%. Moreover, for the 17 snippets that do not answer the question, 12 of them are incomplete, but correct, answers.

From the result of 3-step qualitative analysis, we can see that multiple-line snippets can best answer the questions. This subset contains 40,245 runnable snippets (29.8% of all runnable Python snippets).

2.5 Google Search Results

In this section, we explore the overlap between Google search results and the usable Python snippets. Specifically, we check if the top results from Google for several queries contain parsable or runnable snippets, as well as these snippets' overall quality.

The methodology was as follows. We selected 100 programming related questions from SO's highest voted questions about Python, and use them as queries using the Google search API. We add the constraint "site:stackoverflow.com" and the keyword "Python" in the in the queries. Moreover, because our database was downloaded in April 2014, we also add a date range restriction.

The accepted answers' usability rates of the Top 1 and Top 10 results from Google are shown in Table 2.6. They are high. As described in Section 2.3.2, we had found that the usability

Table 2.6: Usability Rates of Top Results from Google

	Parsable	Runnable
Top 1	78.1%	30.8%
Top 10	77.9%	29.3%

Search Query

```
Check if a given key already exists in a dictionary
```

URL of One Search Result

```
http://stackoverflow.com/questions/17308754/
python-how-to-check-if-keys-exists-and-retrieve-
value-from-dictionary-in-descen/17308754#17308754
```

Snippet of Accepted Answer

```
value = None
for key in keySet:
    if key in myDict:
        value = myDict[key]
        break
```

Runtime Error

```
name 'keySet' is not defined
```

Figure 2.14: Example of Google Search Result

rates of all the Python snippets in SO are 76% parsable and 25% runnable. The top results on 100 queries to Google on the same SO data have usability rates above those averages. Moreover, the Top 1 results have an even higher usability rate than the Top 10 results.

Also, we find that 33.7% of Top 1 results and 32.5% of Top 10 results are multiple line snippets. Both higher than the average of 30%. So, both from our usability perspective and qualitative analysis perspective, the Google Top 10 search results are better than average, and the Top 1 results are the best.

From the results above, we can also see that the Google top results have a low runnable rate, although higher than average. The main problems encountered in the parsable but not runnable snippets from Google results are those already described for the entire SO snippets (see Section 2.3.3). Specifically, the majority of them suffered from undefined names or modules. An example of a parsable but not runnable Google search result to the query "Check if a given key already exists in a dictionary" is shown in Figure 2.14.

Expecting snippets to be runnable as-is may be too strong of a constraint. Parsable snippets seem to be a much more fertile groud as the base for future automatic code generation.

Given our analysis of the causes of runtime errors, it seems it should be possible to repair a large percentage of them automatically. For example for Python, missing symbol names often indicate a piece of information that needs to come from elsewhere – another snippet, or some default initialization.

Note that we used the questions as-is as queries for Google; not surprisingly, Google always returned those SO questions as the Top 10 hits in each query. Out of the 100 queries we selected, 85 original ones were returned as the first hit by Google. Although 15 original links were not ranked as Top 1, 12 of them were in Top 10. The reason for them not being the first one is that Google seems to have a special heuristic to dealing with "daterange" restrictions. If we remove the "daterange" restriction in our search query, the original ones will appear in Top 1. However, 3 out of 100 queries were not in Top 10 list by Google. We looked at these three cases: one is because of the date range restriction, the second one is because it is a new query out of our date range, and the last one seems to genuinely be because of Google ranking algorithms.

The very high hit rate and, in particular, the top 1 results, confirm Google's efficiency in retrieving relevant information from the Web, something that our work leverages, by design. However, the usability rates on the top hits were encouragingly high, and that is orthogonal to Google's efficiency in finding the most relevant results. These higher-than-average usability rates may be because we used the most popular queries; users of SO value complete answers that have good code snippets, so it is not surprising that the most popular queries have snippets that are better than average.

In general, users search using words that are not exactly the same as the words in the SO questions, so the best snippet of code for their needs may not be in the first position; but, as it is usually the case with Google, it is likely in the top 10 positions. The usability of the snippets in the top 10 positions were not as high, but they were still very high (78% parsable, 29% runnable), and above average of the entire set of Python snippets.

The Google search results over SO snippets are very encouraging. They show that it is possible to go from informal queries in natural language to relatively usable, and correct, code in a large percentage of cases, opening the door to the old idea of programming environments that “do what I mean” [127]. This is possible now due to the emergence of very large mixed-language knowledge bases such as SO.

2.6 Conclusion

Some of our experiment choices deserve an explanation:

- In SO, the concepts of accepted answer and best answer are different. Accepted answer is the one approved by the questioner, while best answer is voted by all viewers. We chose accepted answer in this work because we believe that in a Question&Answer forum as SO, the questioner the one who has the best judgement of whether the answer solves the problem. However, it is possible that the questioner makes mistakes and that the answer voted the best by other viewers is most usable. In the future we will evaluate the usability of best answers and compare the results with those of presented here.
- Our definition of usability is purely technical, and does not include the concept of usefulness other than indirectly, by the fact that the analyzed snippets are in the accepted answers. It is possible that a snippet that does not parse is more useful than the one that runs; or that a snippet that does not parse or run is still useful to the answer the question. For example, if the question asked in SO is not a specific programming query, people may answer with pseudo code, which is not usable in our case, but may also answer the original question. Those cases, however, will always be out of reach of automatic tools, as they will require many more repairs or even translation from pseudo-code to actual code. As such, this study focused conservatively on those snippets that are part of accepted answers and

that show good potential to being used as-is or with little repairs.

In this paper, we examined the usability of code snippets in Stack Overflow. The purpose of our usability analysis is to understand the extent to which human-written snippets of code in sites like SO could be used as basic blocks for automatic program generation. We analyzed code snippets from all the accepted answers for four popular programming languages. For the two statically-typed, compiled languages, C# and Java, we performed parsing and compilation, and for the two dynamic languages, Python and JavaScript, we performed parsing and running experiments. The results show that usability rates for the two dynamic languages is substantially higher than that of the two statically-typed, compiled languages. Heuristic repairs improved the results for Java, but not for C#. Even after the repairs, the compilable rates for both Java and C# are very low. The results lead us to believe that Python and JavaScript are the best choices for program synthesis explorations.

Usability as-is, however, is not enough to ensure that the snippets have high information quality. Our qualitative analysis on the most usable snippets showed that multiple line snippets have the highest potential to answer the questions. We found 40K+ of these for Python, meaning that there is a good potential for processing them automatically.

Finally, in order to close the circle on our original vision, we investigated the extent to which the top results of queries on SO using Google Web search contain these usable snippets. The results are very encouraging, and show a viable path from informal queries to usable code.

Chapter 3

File-level Duplication Analysis of GitHub

The material in this chapter is part of the following paper, and is included here with permission from ACM.

C. V. Lopes, P. Maj, P. Martins, V. Saini, D. Yang, J. Zitny, H. Sajnani, and J. Vitek. Déjàvu: A Map of Code Duplicates on Github. In proceedings of ACM Programming Languages, Volume 1(OOPSLA), Oct. 2017.

This paper explores the extent of cloning in a popular software repository hosting online service GitHub. The study was conducted for projects written in a set of popular programming languages, namely, Java, Javascript, C/C++, and Python. The study was led by Prof. Cristina Lopes and involves collaborations with Prof. Jan Vitek. The detection of clones in the JavaScript projects and their analysis was led by Prof. Jan Vitek and his students. I contributed to this work by first collecting all data in Java, C/C++, and Python, and then conducted three levels of file-level duplication analysis for Python projects, and finally finished qualitative analysis on most duplicated files in Python and Java. In this chapter, I

only included the parts that I was involved in.

3.1 Introduction

The advent of web-hosted open source repository services such as GitHub, BitBucket and SourceForge have transformed how source code is shared. Creating a project takes almost no effort and is free of cost for small teams working in the open. Over the last two decades, millions of projects have been shared, building up a massive trove of free software. A number of these projects have been widely adopted and are part of our daily software infrastructure. More recently there have been attempts to treat the open source ecosystem as a massive dataset and to mine it in the hopes of finding patterns of interest.

When working with software, one may want to make statements about applicability of, say, a compiler optimization or a static bug finding technique. Intuitively, one would expect that a conclusion based on a software corpus made up of thousands of programs randomly extracted from an Internet archive is more likely to hold than one based on a handful of hand-picked benchmarks such as [26] or [122]. For an example, consider [108] which demonstrated that the design of the Mozilla optimizing compiler was skewed by the lack of representative benchmarks. Looking at small workloads gave a very different picture from what could be gleaned by downloading thousands of websites.

Scaling to large datasets has its challenges. Whereas small datasets can be curated with care, larger code bases are often obtained by random selection. If GitHub has over 4.5 million projects, how does one pick a thousand projects? If statistical reasoning is to be applied, the projects must be independent. Independence of observations is taken for granted in many settings, but with software there are many ways one project can influence another. Influences can originate from the developers on the team, for instance the same people will tend to

write similar code. Even more common are the various means of software reuse. Projects can include other projects. Apache Commons is used in thousands of projects, Oracle's SDK is universally used by any Java project, JQuery by most websites. StackOverflow and other discussion forums encourage the sharing of code snippets. Cut and paste programming where code is lifted from one project and dropped into another is another way to inject dependencies. Lastly, entire files can be copied from one project to the next. Any of these

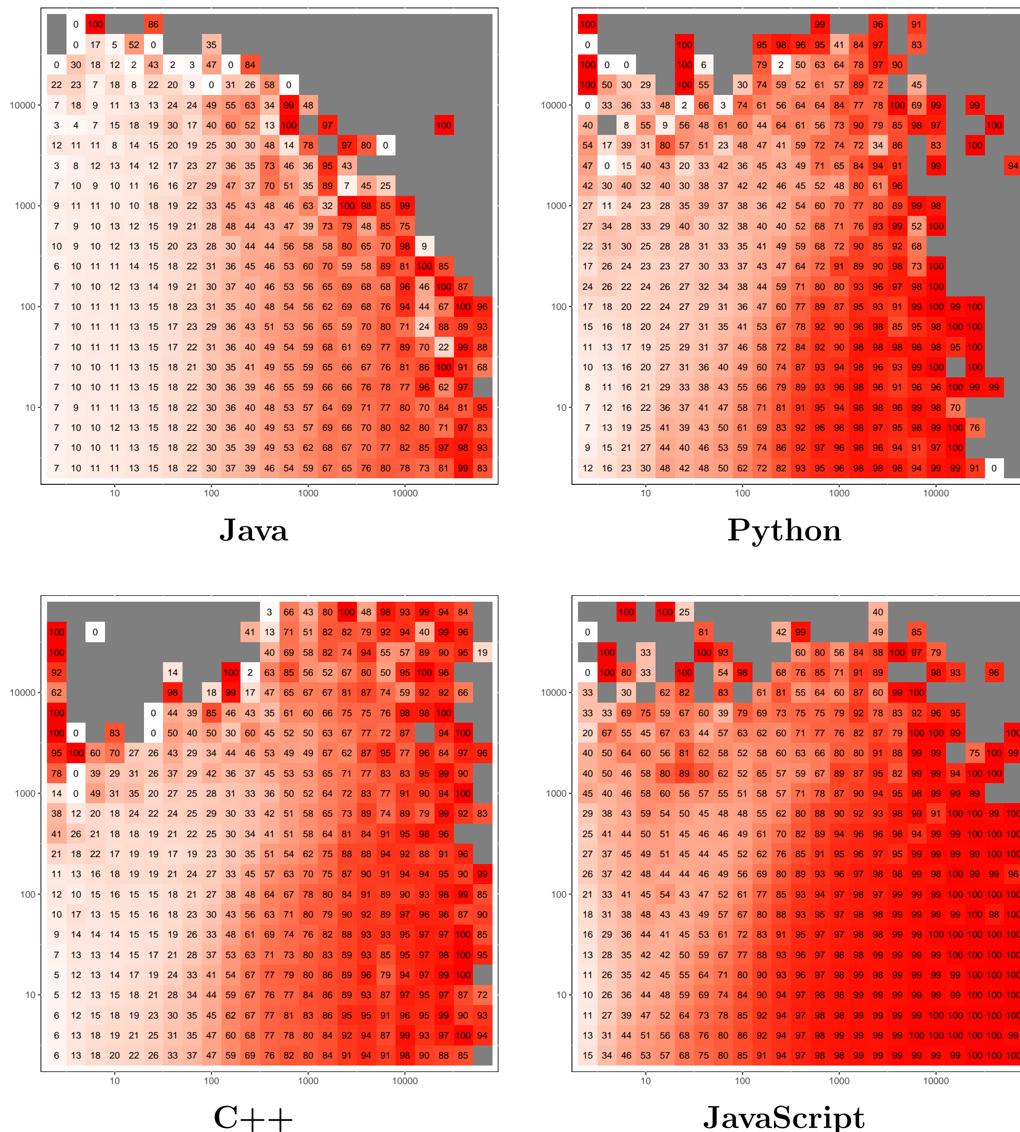


Figure 3.1: Map of code duplication. The y-axis is the number of commits per project, the x-axis is the number of files in a project. The value of each tile is the percentage of duplicated files for all projects in the tile. Darker means more clones.

actions, at scale, may bias results of research.

Several published studies either neglected to account for duplicates, or addressed them before analysis. [33] studied the use of assertions in the top 100 most popular C and C++ projects in GitHub. [106] studied software quality using the top 50 most popular projects in 17 languages. Neither addressed file duplication. Conversely, [61] studied the old "tabs v. spaces" issue in 400K GitHub projects; file duplication was identified as an issue and eliminated before analysis. [39] present a meta-analysis of studies on GitHub projects where trends and problems related to dataset selection are identified.

This paper provides a tool to assist selecting projects from GitHub. DéjàVu is a publicly available index of file-level code duplication. The novelty of our work lies partly in its scale; it is an index of duplication for the entire GitHub repository for four popular languages, Java, C++, Python and JavaScript. Figure 3.1 illustrates the proportion of duplicated files for different project sizes and numbers of commits (section 3.5 explains how these heatmaps were generated). The heatmaps show that as project size increases the proportion of duplicated files also increases. Projects with more commits tend to have fewer project-level clones. Finally JavaScript projects have the most project-level clones, while Java projects have the fewest.

Table 3.1: File-hash duplication in subsets.

	10K Stars	10K Commits
Java	9%	6%
C/C++	41%	51%
Python	28%	44%
JavaScript	44%	66%

The clone map from which the heatmaps were produced is our main contribution. It can be used to understand the similarity relations in samples of projects or to curate samples to reduce duplicates. Consider for instance a subset that focuses on the most active projects, as done in [27], by filtering on the number of stars or commits a project has. For example, the clones for the 10K most popular projects are summarized in Figure 3.1. In Java, this filter is reasonably efficient at reducing the number of clones.

In other languages clones remain prevalent. DéjàVu can be used to curate datasets, i.e. remove projects with too many clones. Besides applicability to research, our results can be used by anyone who needs to host large amounts of source code to avoid storing duplicate files. Our clone map can also be used to improve tooling, e.g. being queried when new files are added to projects to filter duplicates.

At the outset of this work, we were planning to study different granularities of duplication. As the results came in, the staggering rate of file-level duplication drove us to select three simple levels of similarity. A *file hash* gives a measure of file that are copied across projects without changes. A *token hash* captures minor changes in spaces, comments and ordering. Lastly, SourcererCC captures files with 80% token-similarity. This gives an idea of how many files have been edited after cloning. Our choice of languages was driven by the popularity of these languages, and by the fact that two are statically typed and two have no type annotations. This can conceivably lead to differences in the way code is reused. We expected to answer the following questions: How much code cloning is there, how does cloning affect datasets of software written in different languages, and through which processes does duplication come about? This paper describes our methodology, details the corpus that we have selected and gives our answers to these questions. Along with the quantitative analysis, we provide a qualitative analysis of duplicates on a small number of examples.

Artifacts. The lists of clones, code for gathering data, computing clones, data analysis and visualization are at: `http://mondego.ics.uci.edu/projects/dejavu`. Processing was done on a Dell PowerEdge R830 with 56 cores (112 threads) and 256G of RAM. The data took 2 months to download and 6 weeks to process.

3.2 Related Work

Code clone detection techniques have been documented in the literature since the early 90s. Readers interested in a survey of the early work are referred to [72, 110]. There are also benchmarks for assessing the performance of tools [111, 126]. The pipeline we used includes SourcererCC, a token-based code clone detection tool that is freely available and has been compared to other similar tools using those benchmarks [116].[1] SourcererCC is the most scalable tool so far for detecting Type 3 clones. Type 3 clones are syntactically similar code fragments that differ at the statement level. The fragments have statements added/modified/removed with respect to each other.

One of the earliest studies of inter-project cloning, [66] analyzed clones across three different operating systems. They found evidence of about 20% cloning between FreeBSD and NetBSD and less than 1% between Linux and FreeBSD or NetBSD. This is explained by the fact that Linux originated and grew independently. [83] performed an analysis of popular open source projects, including several versions of Unix and several popular packages; 38K projects and 5M files. The concept of duplication there was simply based on file names. Approximately half of the file names were used in more than one project. Furthermore, the study also tried to identify components that were duplicated among projects by detecting directories that share a large fraction of their files. Both [83] and [84] use only a fraction of our dataset and a single similarity metric, as opposed to the 3 metrics we provide.

A few studies have focused on block-level cloning, i.e. portions of code smaller than entire files. [112] analyzed clones in twenty open source C, Java and C# systems. They found 15% of the C files, 46% of the Java files, and 29% of C# files are associated with exact block-level clones. Java had a higher percentage of clones because of accessors methods in Swing. [58] computed block-level clones consisting of at least 15 statements between 22 commonly reused

[1] `http://github.com/Mondego/SourcererCC`

Java frameworks consisting of more than 6 MLOC and 20 open source Java projects. They did not find any clones for 11 projects. For 5 projects, they found cloning to be below 1% and for the remaining 4, they found up to 10% cloning. These two studies give conflicting accounts of block-level code duplication.

Closer to our study, an analysis of file-level code cloning on Java projects is presented by [98]. This work, analyzed 13K Java projects with close to 2M files. The authors created a system that merges various clone detection techniques with various degrees of confidence, starting on the highest: MD5 hashes; name equivalence through Java's full-qualified names. They report 5.2% file-hash duplication, considerably lower than what we found. Our corpus is three orders of magnitude larger than Ossher's. Furthermore, intra-project duplication meant to deal with versioning was excluded. They looked at subversion, which may have different practices than git, especially related to versioning. We speculate that the practice of copying source code files in open source has become more pervasive since that study was made, and that sites like GitHub simplify copying files among projects, but we haven't reanalyzed the dataset as it is not relevant to the DéjàVu map.

Over the past few years, open source repositories have turned out to be useful to validate beliefs about software development and software engineering in general. The richness of the data and the potential insights that it represents has created an entire community of researchers. [71] used 50K GitHub repositories to investigate the correlation between the presence of test cases and various project development characteristics, including the lines of code and the size of development teams. They removed toy projects and included famous projects such as JQuery and Rails in their dataset. [140] study how licensing usage and adoption changes over a period of time on 51K repositories. They choose repositories that (i) were not forks; and (ii) had at least one star. [27] analyze 2.5K repositories to investigate the factors that impact their popularity, including the identification of the major patterns that can be used to describe popularity trends.

The software engineering research community is increasingly examining large number of projects to test hypotheses or derive new knowledge about the software development process. However, as [88] point out, more is not necessarily better, and selection of projects plays an important role – more so now than ever, since anyone can create a repository for any purpose at no cost. Thus, the quality of data gathered from these software repositories might be questionable. For example, as we also found out, repositories often contain school assignments, copies of other repositories, images and text files without any source code. [65] manually analyzed a sample of 434 GitHub repositories and found that approximately 37% of them were not used for software development. As a result, researchers have spent significant effort into collecting, curating, and analyzing data from open source projects around the world. Flossmetrics [56] and Sourcerer [97] collect data and provide statistics. [44] have curated a large number of Java repositories and provide a domain specific language to help researchers mine data about software repositories. Similarly [25] have created Orion, a prototype for enabling unified search to retrieve projects using complex search queries linking different artifacts of software development, such as source code, version control metadata, bug tracker tickets, developer activities and interactions extracted from hosting platform. Black Duck Open Hub (`www.openhub.net`) is a public directory of free and open source software, offering analytics and search services for discovering, evaluating, tracking, and comparing open source code and projects. It analyzes both the code's history and ongoing updates to provide reports about the composition and activity of project code bases. These platforms are useful for researchers to filter out repositories that are interesting to study a given phenomenon by providing various filters. While these filters are useful to validate the integrity of the data to some extent, certain subtle factors when unaccounted for can heavily impact the validity of the study. Code duplication is one such factor. For example, if the dataset consists of projects that have hundreds and thousands of duplicate projects that are part of the same dataset, the overall lack of diversity in the dataset might lead to incorrect observations, as pointed out by [88].

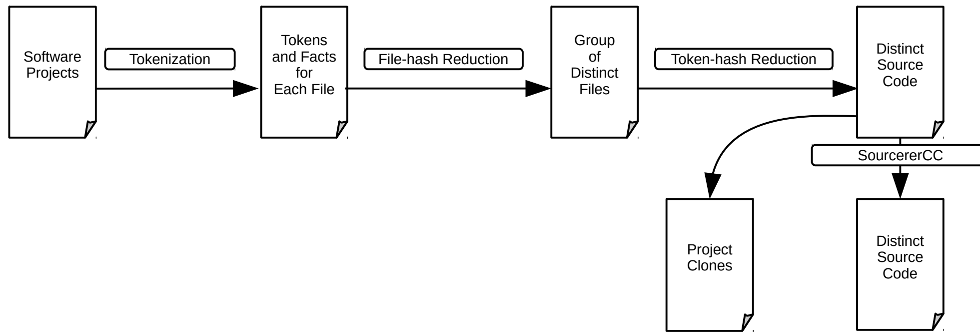


Figure 3.2: Analysis pipeline.

3.3 Analysis Pipeline

Our analysis pipeline is outlined in Figure 7.2. The pipeline starts with local copies of the projects that constitute our corpus. From here, code files are scanned for fact extraction and tokenization. Two of the facts are the hashes of the files and the hashes of the tokens of the files. File hashes identify exact duplicates; token hashes allow catch clones up with minor differences. While permutations of same tokens may have the same hash, they are unlikely. Clones are dominated by exact copies, and we did not observe any such collision in randomly sampled pairs. Files with distinct token hashes are used as input to the near-miss clone detection tool, SourcererCC. While our JavaScript pipeline was developed independently, data formats, database schema and analysis scripts are identical.

3.3.1 Tokenization

Tokenization transforms a file into a "bag of words," where occurrences of each word are recorded. Consider, for instance, the Java program:

```
package foo;
public class Foo { // Example Class
private int x;
public Foo(int x) { this.x = x; }
private void print() { System.out.println("Number: " + x) }
public static void main() { new FooNumber(4).print(); } }
```

Projects

<u>Project Id</u>	Project Path	GitHub URL

Files

<u>File Id</u>	Project Id	Relative Path	Relative URL	File Hash

Stats

<u>File Hash</u>	Bytes	Lines	LOC	SLOC	Tokens	...

...	Unique Tokens	Token Hash

Tokenization removes comments, white space, and terminals. Tokens are grouped by frequency, generating:

```
Java Foo:[(package,1),(foo,1),(public,3),(class,1),(Foo,2),(private,2),(int,2),(x,5),
(this,1),(void,2),(print,2),(System,1),(out,1),(println,1),(Number,1),(static,1),
(main,1),(new,1),(FooNumber,1),(4,1)]
```

The tokens `package` and `foo` appear once, `public` appears three times, etc. The order is not relevant. During tokenization we also extract additional information: (1) *file hash* – the MD5 hash of the entire string that composes the input file; (2) *token hash* – the MD5 hash of the string that constitutes the tokenized output; (3) *size* in bytes; (4) *number of lines*; (5) *number of lines of code* without blanks; (6) *number of lines of source* without comments; (7) *number of tokens*; and (8) *number of unique tokens*. The tokenized input is used both to build a relational database and as input to SourcererCC. The use of MD5 (or any hashing algorithm) runs the risk of collisions, given the size of our data they are unlikely to skew the results.

3.3.2 Database

The data extracted by the tokenizer is imported into a MySQL database. The table `Projects` contains a list of projects, with a unique identifier, a path in our local corpus and the project's

URL. `Files` contains a unique id for a file, the id of the project the file came from, the relative paths and URLs of the file and the file hash. The statistics for each file are stored in the table `Stats`, which contains the information extracted by the tokenizer. The tokens themselves are not imported. The `Stats` table has the file hash as unique key. With this, we get an immediate reduction from files to hash-distinct files. Two files with distinct file hashes may produce the exact same tokens, and, therefore the same token hash. This could happen when the code of one file is a permutation of another. The converse does not hold: files with distinct token hashes must have come from files with distinct file hashes. For source code analysis, file hashes are not necessarily the best indicators of code duplication; token hashes are more robust to small perturbations. We use primarily token hashes in our analysis.

3.3.3 SourcererCC

The concept of inexact code similarity has been studied in the code cloning literature. Blocks of code that are similar are called near-miss clones, or near-duplication [37]. SourcererCC estimates the amount of near-duplication in GitHub with a "bag of words" model for source code rather than more sophisticated structure-aware clone detection methods. It has been shown to have good precision and recall, comparable to more sophisticated tools [116]. Its input consists of non-empty files with distinct token hashes. SourcererCC finds clone pairs between these files at a given level of similarity. We have selected 80% similarity as this has given good empirical results. Ideally one could imagine varying the level of similarity and reporting a range of results. But this would be computationally expensive and, given the relatively low numbers of near-miss clones, would not affect our results.

Table 3.2: GitHub Corpus.

		Java	C++	Python	JavaScript
Counts	# projects (total)	3,506,219	1,130,879	2,340,845	4,479,173
	# projects (non-fork)	1,859,001	554,008	1,096,246	2,011,875
	# projects (downloaded)	1,481,468	369,440	909,290	1,778,679
	# projects (analyzed)	1,481,468	364,155	893,197	1,755,618
	# files (analyzed)	72,880,615	61,647,575	31,602,780	261,676,091
Medians	Files/project	9 ($\sigma = 600$)	11 ($\sigma = 1304$)	4 ($\sigma = 501$)	6 ($\sigma = 1335$)
	SLOC/file	41 ($\sigma = 552$)	55 ($\sigma = 2019$)	46 ($\sigma = 2196$)	28 ($\sigma = 2736$)
	Stars/project	0 ($\sigma = 71$)	0 ($\sigma = 119$)	0 ($\sigma = 99$)	0 ($\sigma = 324$)
	Commits/project	4 ($\sigma = 336$)	6 ($\sigma = 1493$)	6 ($\sigma = 542$)	6 ($\sigma = 275$)

3.4 Corpus

The GitHub projects were downloaded using the GHTorrent database and network [57] which contains meta-data such as number of stars, commits, committers, whether projects are forks, main programming language, date of creation, etc., as well as download links. While convenient, GHTorrent has errors: 1.6% of the projects were replicated entries with the same URL; only the youngest of these was kept for the analysis.

Table 3.2 gives the size of the different language corpora. We skipped forked projects as forks contain a large amount of code from the original projects, retaining those would skew our findings. Downloading the projects was the most time-consuming step. The order of downloads followed the GHTorrent projects table, which seems to be roughly chronological. Some of the URLs failed to produce valid content. This happened in two cases: when the projects had been deleted, or marked private, and when development for the project happens in branches other than master. Thus, the number of downloaded projects was smaller than the number of URLs in GHTorrent. For each language, the files analyzed were files whose extensions represent source code in the target languages. For Java: `.java`; for Python: `.py`; for JavaScript: `.js`, for C/C++: `.cpp .hpp .HPP .c .h .C .cc .CPP .c++` and `.cp`. Some projects did not have any source code with the expected extension, they were

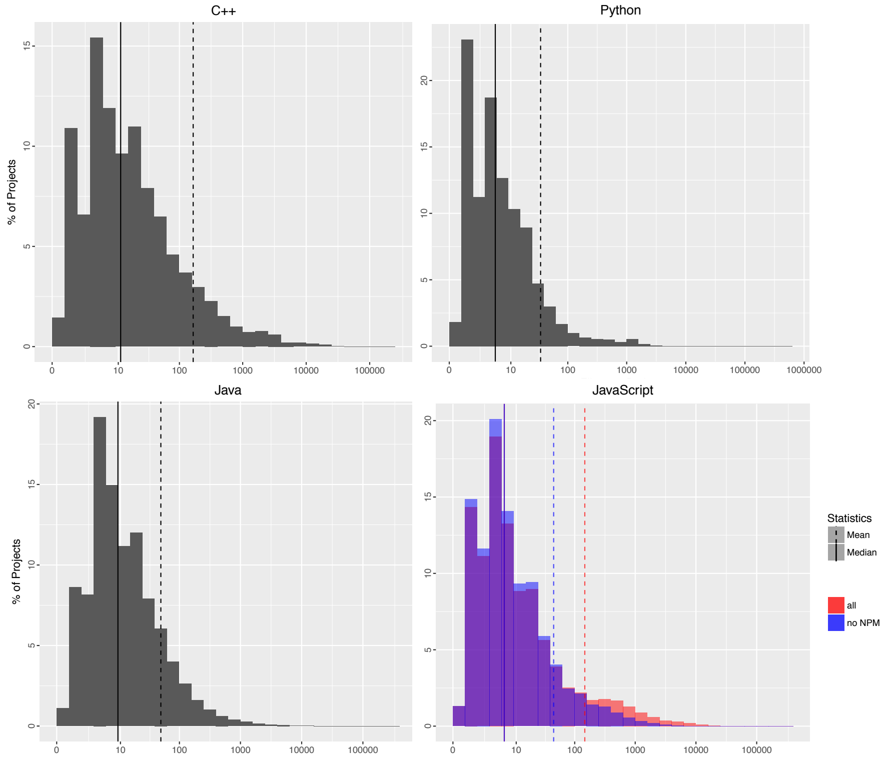


Figure 3.3: Files per project.

excluded.

The medians in Table 3.2 give additional properties of the corpus, namely the number of files per (non-empty) project, the number of Source Lines of Code (SLOC) per file, the number of stars and the number of commits of the projects. In terms of files per project, Python and JavaScript projects tend to be smaller than Java and C++ projects. C++ files are considerably larger than any others, and JavaScript files are considerably smaller. None of these numbers is surprising. They all confirm the general impression that a large number of projects hosted in GitHub are small, not very active, and not very popular. Figures 3.3 and 3.4 illustrate the basic size-related properties of the projects we analyzed, namely the distribution of files per project and the distribution of Source Lines of Code (SLOC) per file. For JavaScript we give data with and without NPM (it is a cause of a large number of clones). Without NPM means that we ignored files downloaded by the Node Package Manager.

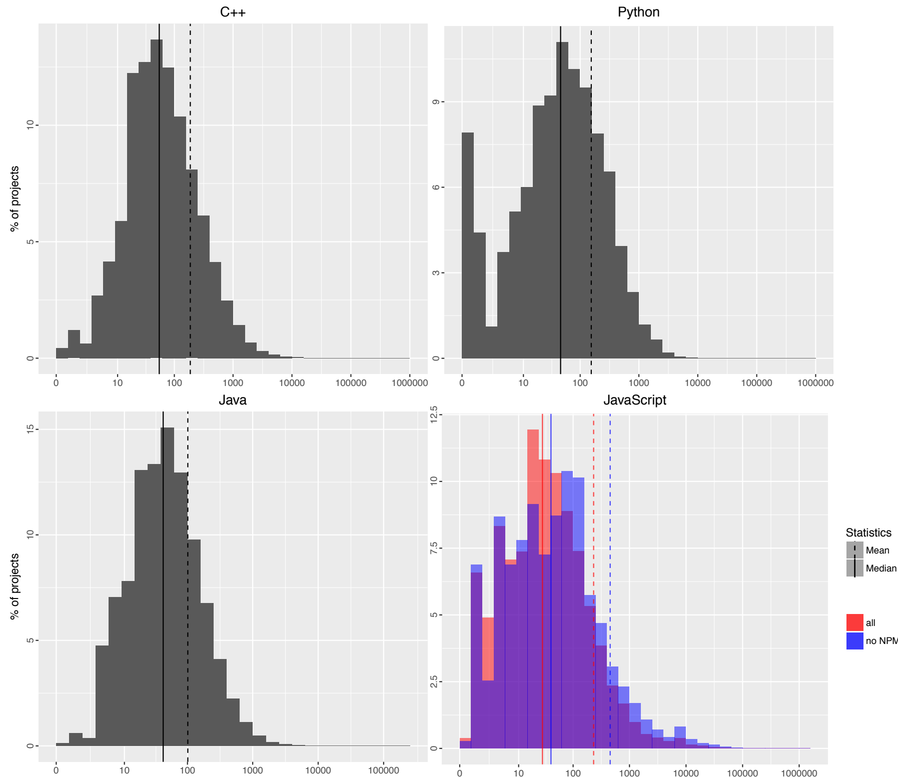


Figure 3.4: SLOC per file.

3.5 Quantitative Analysis

We present analyses of the data at two levels of detail: file and project level. This section focuses exclusively on quantitative analysis; the next section delves deeper into qualitative observations.

3.5.1 File-Level Analysis

Table 3.3 shows a summary of the findings for files. "SCC dup files" is the number of files, out of the distinct token-hash files, that SourcererCC has identified as clones; similarly, "SCC unique files" is the number of files for which no clones were detected. Figure 3.5 (top row) charts the numbers in Table 3.3. The duplicated files (dark grey) are the files that are duplicate of at least one of the distinct token-hash files (light grey); further, the distinct

token-hash files are split between those for which SourcererCC found at least one similar file (cloned files, grey) and those for which SourcererCC did not find any similar file (unique files, in white).

These numbers show a considerable amount of code duplication, both exact copies of the files (file hashes), exact copies of the files' tokens (token hashes), and near-duplicates of files (SourcererCC). The amount of duplication varies with the language: the JavaScript ecosystem contains the largest amount of duplication, with 94% of files being file-hash clones of the other 6%; the Java ecosystem contains the smallest amount, but even for Java, 40% of the files are duplicates; the C++ and Python ecosystems have 73% and 71% copies, respectively. As for near-duplicates, Java contains the largest percentage: 46% of the files are near-duplicate clones. The ratio of near-miss clones is 43% for Java, 39% for JavaScript, and 32% for Python.

The heatmaps (Figure 3.1) shown in the beginning of the paper were produced using the number of commits shown in Table 3.2, the number of files in each project, and the file hashes. The heat intensity corresponds to the ratio of file hashes clones over total files for each cell.

Duplication can come in many flavors. Specifically, it could be evenly or unevenly distributed among all token hashes. We found these distributions to be highly skewed towards small groups of files. In Java 1.5M groups of files with the same token-hash have either 2 or 3 files in them; the number of token hash-equal groups with more than 100 files is minuscule. The

Table 3.3: File-Level Duplication.

	Java	C++	Python	JavaScript
Total files	72,880,615	61,647,575	31,602,780	261,676,091
File hashes	43,713,084 (60%)	16,384,801 (27%)	9,157,622 (29%)	15,611,029 (6%)
Token hashes	40,786,858 (56%)	14,425,319 (23%)	8,620,326 (27%)	13,587,850 (5%)
SCC dup files	18,701,593 (26%)	6,200,301 (10%)	2,732,747 (9%)	5,245,470 (2%)
SCC unique files	22,085,265 (30%)	8,225,018 (13%)	5,887,579 (19%)	8,342,380 (3%)

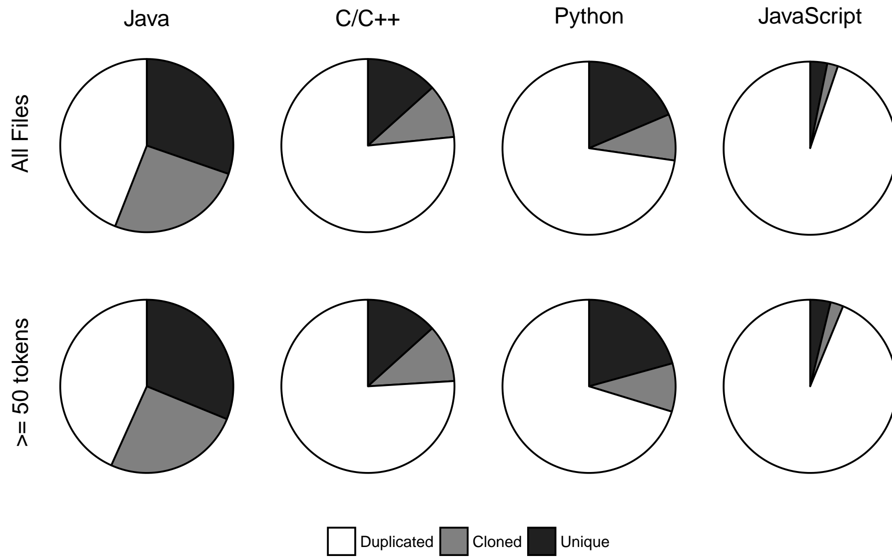


Figure 3.5: File-level duplication for entire dataset and excluding small files.

same observation holds for the other languages. Another interesting piece of information about clone groups is given by the largest extreme. In Python, the largest group of file-hash clones has over 2.5M files. In Java, the largest group of SourcererCC clones has over 65K files. In the next section we show which files these are.

3.5.2 File-Level Analysis Excluding Small Files

One observation that emerged immediately from all the language ecosystems was that the most duplicated file is the empty file – a file with no content, and size 0. In the Python corpus alone, there are close to 2.2M occurrences of this trivial file, and in the JavaScript corpus there are 986K occurrences of that same file. Another frequently occurring trivial file in all ecosystems is a file with 1 empty line. Indeed, a common pattern that emerged was that the most duplicated files tend to be very small. Once we detected that, we redid the analysis excluding small files. Specifically, we excluded all files with less than 50 tokens.[2] Table 3.4 and Figure 3.5 (bottom row) show the results.

[2] This threshold is arbitrary. It is based on our observations of small files; other values can be used.

Although the absolute number of files and hashes change significantly, the changes in ratios of the hashes and SCC results are small. When they are noticeable, they show that there is slightly less duplication in this dataset than in the entire dataset. Comparing Table 3.4 with Table 3.3 shows that small files account for a slightly higher presence of duplication, but not that much higher than the rest of the corpus.

3.6 Mixed Method Analysis

The numbers presented in the previous section portray an image of GitHub not seen before. However, that quantitative analysis opens more questions. What files are being copied around, and why? What explains the differences between the language ecosystems? Why is the JavaScript ecosystem so much off the charts in terms of duplication? In order to answer these kinds of questions, we delve deeper into the data.

With so much data, our first heuristic was size. As seen in the previous section we noticed that the empty file was the most duplicated file in the entire corpus, among all languages. We also noticed that the top duplicated files tended to be very small and relatively generic. Although an intriguing finding, very small, generic files hardly provide any insightful information about the practice of code duplication. What about the non-trivial files that are heavily duplicated? What are they?

Table 3.4: File-level duplication excluding small files.

	Java	C++	Python	JavaScript
# of files	57,240,552	49,507,006	23,382,050	162,136,892
% of corpus	79%	80%	74%	62%
File hashes	34,617,736 (60%)	13,401,948 (27%)	7,267,097 (31%)	11,444,667 (7%)
Token hashes	32,473,052 (58%)	11,893,435 (24%)	6,949,894 (30%)	10,074,582 (6%)
SCC dup files	14,626,434 (26%)	5,297,028 (10%)	2,105,769 (9%)	3,896,989 (2%)
SCC unique files	17,848,618 (31%)	6,596,407 (13%)	4,844,125 (21%)	6,177,593 (4%)

This section presents observations emerging from looking at specific files and projects using mixed methods. We divide the section into four parts: (1) an analysis of each language ecosystems looking for the most duplicated files in general; (2) file duplication at different levels (file hashes, token hashes and near duplicates with SourcererCC); (3) the most reappropriated projects in the four ecosystems; and (4) an in-depth analysis of the JavaScript ecosystem.

3.6.1 Most Duplicated Non-Trivial Files

As stated above, we wanted to find out if the size of the files had an effect on their duplication. For example, are small files copy-pasted from StackOverflow or online tutorials and blogs, and large files from well-known supporting libraries? In order to make sense of so much data, we needed to sample it first, so that interesting hypotheses could emerge, and/or we could find counter-examples that contradicted our initial expectations. This is territory of qualitative and mixed methods [41].

3.6.1.1 Methodology

We used a mixed method approach consisting of qualitative and quantitative elements. Based on our quantitative analysis, we hypothesized that size of the files, and whether the duplication was exact or token-based, might have an effect on the nature of duplication; for example, the empty file certainly is not being copy-pasted from one project to another, it simply is created in many projects, for a variety of reasons. Maybe we could see patterns emerge for files of different sizes. The following describes our methodology:

- **Quantitative Elements.** We split files according to the percentiles of the number of tokens per file within each language corpus, and create bins representing the ranges 20%-

Table 3.5: Number of tokens per file within certain percentiles of the distribution of file size.

		20%-30%	45%-55%	70%-80%	90%+
Tokens	Java	46-71	120-167	279-419	751+
	C/C++	50-77	138-199	372-623	1284+
	Python	29-65	149-236	477-795	1596+
	JavaScript	19-32	68-114	238-431	1127+
Files	Java	7,670,926 (11%)	7,523,679 (10%)	7,335,067 (10%)	7,298,767 (10%)
	C/C++	6,381,850 (10%)	6,228,550 (10%)	6,204,943 (10%)	6,167,647 (10%)
	Python	3,282,957 (10%)	3,205,337 (10%)	3,169,316 (10%)	3,161,325 (10%)
	JavaSript	28,257,319 (11%)	27,306,195 (10%)	26,326,975 (10%)	26,134,513 (10%)

30% (small), 45%-55% (medium), 70%-80% (large), and greater than 90% (very large). So, the 45%-55% bin contains files that are between the 45% percentile and the 55% percentile on the number of tokens per file of a certain language. The number of tokens for the bins can be seen in Table 3.5. For example in Java, the first bin includes files containing 47 to 72 tokens, and so on. The gaps between these percentiles (for example, no file is observed between the 30% and the 45% percentile) ensure buffer zones that are large enough to isolate the differently-sized files, should differences in their characteristics be observed. For each of these bins, we analyzed the top 20 most cloned files; this grouping was performed twice, using file hashes and token hashes, and this was done for all the languages. In total, for each language, 80 files were analyzed.

- **Qualitative Elements.** Looking at names of most popular files, a first observation was that many of these files came from popular libraries and frameworks, like Apache Cordova. This hinted at the possibility that the origin of file duplication was in well-known, popular libraries copied in many projects; a qualitative analysis of file duplication was better understood from this perspective. Therefore, each file was observed from the perspective of the path relative to the project where it resides, and was then *hand coded* for its origin.[3] For example, `project_name/src/external/com/http-lib/src/file.java` was considered to be part of the external library `http-lib`. Each folder assumed to

[3]For a good tutorial on coding, see [117]

represent an external library was matched with an existing homepage for the library, if we could find it using Google. Continuing the running example, `http-lib` was only flagged as an external dependency if there was a clear pointer online for a Java library with that name. In some cases, the path name was harder to interpret, for example: `p` roject_name/external/include/internal/ftobjs.h. In those cases, we searched Google for the last part of the path in order to find the origin (in this particular case, we searched `i` nclude/internal/ftobjs.h). For JavaScript the situation was often simpler: many of the files came from NPM modules, in which case the module name was obvious from the file's location. Some of the files were also minified versions of libraries, in which case the name of the file gave the library name, often with its version (e.g. `jquery-3.2.1.min`). Using these methods, we were able to trace the origins of all the 320 files.

3.6.1.2 Observations

Contrary to our original expectation, we did not find any differences in the nature of file duplication related to either size of the files, similarity metric, or language in the 320 samples we inspected. We also didn't find any StackOverflow or tutorial files in these samples. Moreover, the results for these files show a pattern that crosses all of those dimensions: the most duplicated files in all ecosystems come from a few well-known libraries and frameworks. The Java files were dominated by the ActionBarSherlock and Cordova. C/C++ was dominated by boost and freetype, and JavaScript was dominated by files from various NPM packages, only 2 cases were from jQuery library. For Python, the origins of file cloning for the 80 files sampled were more diverse, along 6 or 7 common frameworks.[4]

Because the JavaScript sample was so heavily (78 out of 80) dominated by Node packages, we have performed the same analysis again, this time excluding the Node files. This uncov-

[4] The very small number of libraries and frameworks found in these samples is a consequence of having sampled only 80 files per language, and the most duplicated ones. Many of the files had the same origin, because those original libraries consist of several files.

ered jQuery in its various versions and parts accounting for more than half of the sample (43), followed from a distance by other popular frameworks such as Twitter Bootstrap (12), Angular (7), reveal (4). Language tools such as modernizr, prettify, HTML5Shiv and others were present. We attribute this greater diversity to the fact that to keep connections small, many libraries are distributed as a single file. It is also a testament to the popularity of jQuery which still managed to occupy half of the list.

The presence of external libraries within the projects' source code shows a form of dependency management that occurs across languages, namely, some dependencies are source-copied to the projects and committed to the projects' repositories, independent of being installed through a package manager or not. Whether this is due to personal preference, operational necessity, or simple practicality cannot be inferred from our data.

Another interesting observation was the proliferation of libraries for being themselves source-included in other widely-duplicated libraries. Take Cordova, a common duplicated presence within the Java ecosystem. Cordova includes the source of okhttp, another common origin of duplication. Similarly, within C/C++, freetype2 was disseminated in great part with the help of another highly dispersed supporting framework, cocos2d. This not only exacerbates the problem, but provides a clear picture of the tangled hierarchical reliance that exists in modern software, and that sometimes is source-included rather than being installed via a package manager.

3.6.2 File Duplication at Different Levels

In this section, we look in greater detail at the duplication in the three levels reported: file hashes, token hashes and SCC clones:

3.6.2.1 File Hashes

Top cloned files of various sizes were already analyzed in 3.6.1. To complement, we have also investigated mostly cloned non-trivial files across all sizes to make sure no interesting files slipped between the bins, but we did not find any new information. Instead we tried to give more precise answer to question which files get cloned most often. Our assumption was that the smaller the file, the more likely it is to be copied. Figure 3.6 shows our findings. Each file hash is classified by number of copies of the file (horizontal axis) and by size of the file in bytes (vertical axis). Furthermore, we have binned the data into 100x100 bins and we have a logarithmic scale on both axes, which forms the artefacts towards the axes of the graph. The darker the particular bin, the more file hashes it contains. The graphs show that while it is indeed smaller files that get copied most often, with the exception of extremely small outliers (trivial files, such as the empty file), the largest duplication groups can be found for files with sizes in thousands of bytes, with maximum sizes of the clone groups gradually lowering for either larger, or smaller files.

3.6.2.2 Token Hashes

For a glimpse of the distribution of token hashes, we have investigated the relations between number of files within a token hash group and number of file hashes (i.e. different files). These findings are summarized in Figure 3.7. The outlier in the top-right corner of each graph is the empty file. The number of different empty files is explained by the fact that when using token hash, any file that does not have any language tokens in it is considered empty. Given the multitude of sizes observed within token hash groups, the next step was to analyze the actual difference in sizes within the groups. The results shown in Figure 3.8 summarize our findings. As expected, for all four languages the empty file again showed very close to the top. For Java, the biggest empty file was 24.3MB and contains a huge number of

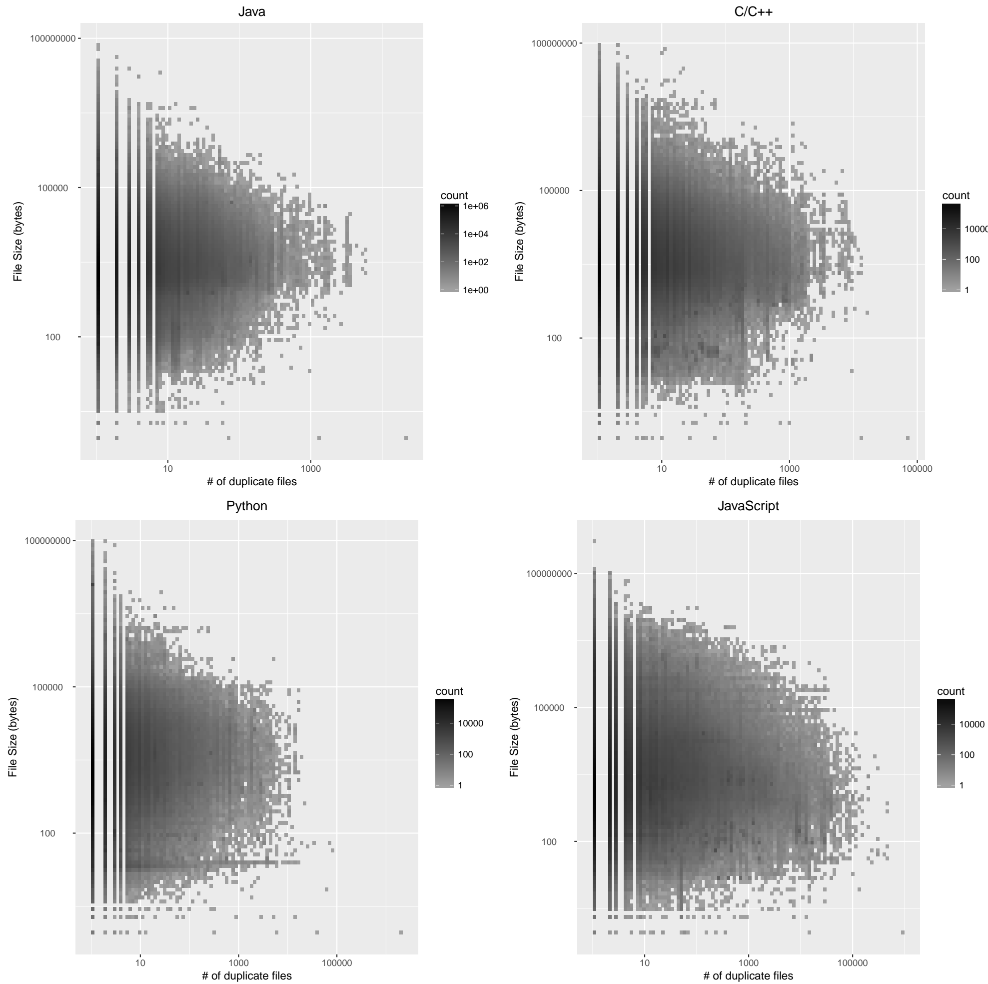


Figure 3.6: Distribution of file-hash clones.

comments as a compiler test. For C/C++ the empty files has the second largest difference and consists of a comment with ASCII art. Python's empty file was a JSON dump on a single line, which was commented, and finally for JavaScript the largest empty file consisted of thousands of repetitions of an identical comment line, totaling 36MB.

More interesting than largest empty files is the answer to the question: What other, non-trivial files display the greatest difference between sizes in the same group. Interestingly, the answer is slightly different for each language: for Java, the greatest size differences exist for binary files disguised as java files. In these files, very few tokens were identified by the tokenizer and therefore two unrelated binary files were grouped into a single token group with

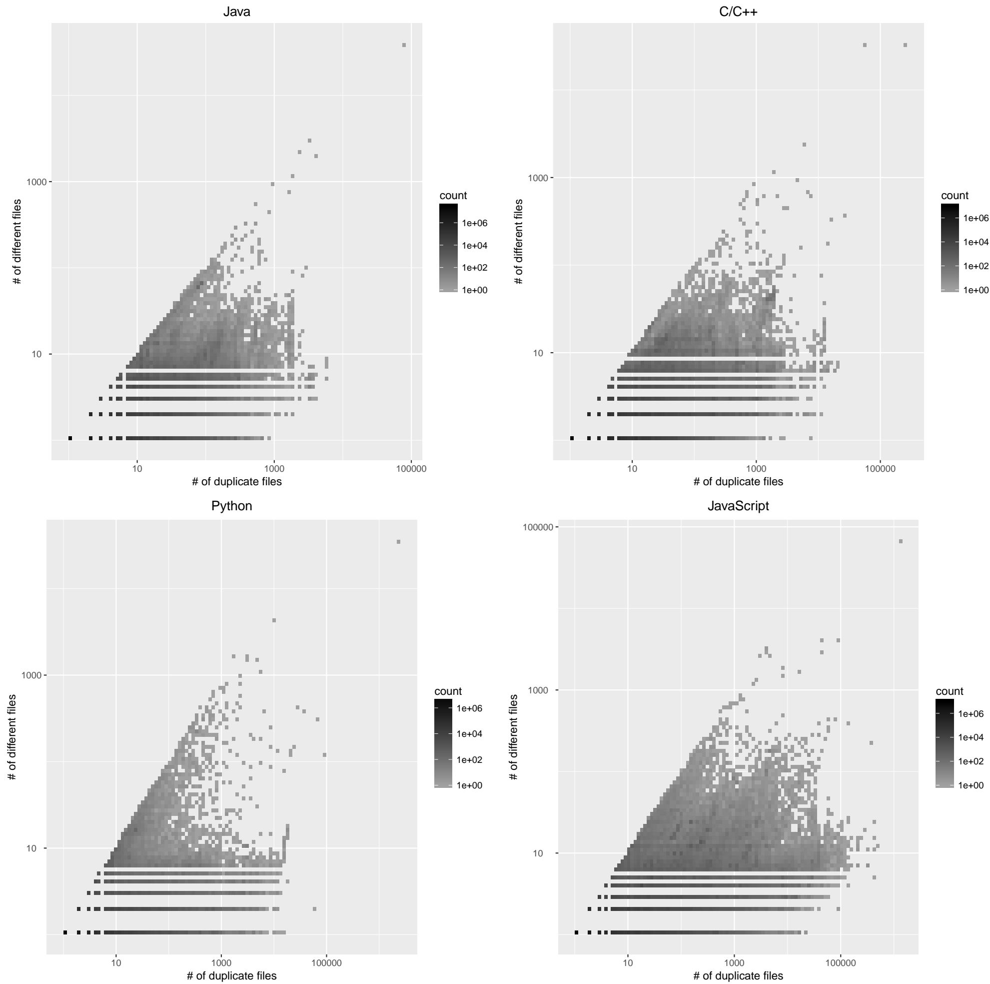


Figure 3.7: Distribution of token-hash clones.

a small number of very different files. For C/C++ often, we have found source codes with and without hundreds of KB of comments as members of the same groups. An outlier was a file with excessive white-spaces at each line (2.42MB difference). In Python, formatting was most often the cause: a single file multiplied its size 10 times by switching from tabs to 8 spaces. For JavaScript, we observed minified and non-minified versions. Sometimes the files were false positives because complex Javascript regular expressions were treated as comments by the simple cross-language parser.

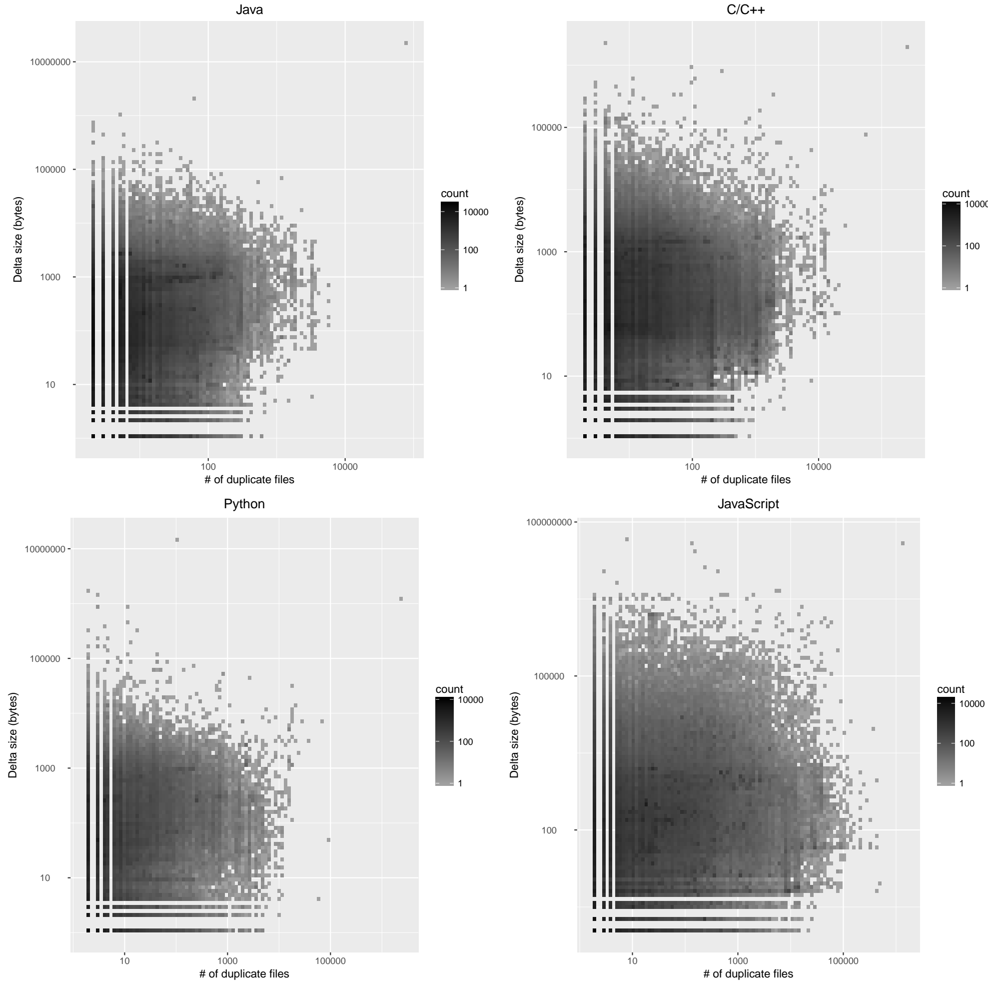


Figure 3.8: Δ of file sizes in token hash groups.

3.6.2.3 SourcererCC Duplicates

For SourcererCC, we randomly selected 20 clone pairs and we categorized them into three categories: i) *intentional copy-paste clones*; ii) *unintentional accidental clones*; and iii) *auto-generated clones*. It is interesting to note that the clones in categories ii) and iii) are both unavoidable and are created because of the use of the popular frameworks.

Java We have categorized 30% (6 out of 20) of the clone pairs into the *intentional copy-paste clones* category. It included instances of both inter-project and intra-project clones. Intra-project clones were created to test/implement functionalities that are similar while

keeping them isolated and easy to maintain. Inter-project clones seemed to come from projects that look like class projects for a university course and from situations where one project was almost entirely copy-pasted into the other project. We found 2 instances of *unintentional cloning*, both inter-project. The files in such clone pairs implement a lot of similar boilerplate code necessary to create an Android activity class. We categorized the majority (12 out of 20) of the clone pairs into the *auto-generated clones* category. The files in this category are automatically generated from the frameworks like Apache Axis (6 pairs), Android (2 pairs), and Java Architecture for XML Binding (4 pairs). The unintentional and auto-generated clones together constitute 70% of the sample.

C/C++ The sample was dominated by *intentional copy-paste clones* (70%, 12 pairs). The origin for these file clone pairs seems to be the same, independent of these being inter of intra-project clones, and relates to the reuse of certain pieces of source code after which they suffer small modification to cope with different setups or support different frameworks. Five pairs were classified as *unintentional cloning*. They represented educational situations (one file was composed in its large part by the skeleton of a problem, and the difference between the files clones was the small piece of code that implements the solution). Two different versions of the same file were also found (libpng 1.0.9 vs. libpng 1.2.30). Files from two projects sharing a common ancestor (bitcoin vs dotcoin) were also observed. The *auto-generated clones* were present in three pairs, 2 of them from the Meta-Object compiler.[5] The unintentional and auto-generated clones accounted for 40% of the sample.

Python The sample was dominated by uses of the Django framework (17 pairs), all variants of auto generated code to initialize a Djagno application. We classified them as *auto-generated clones*. Two pairs were *intentional copy-paste clones* intra-project copy-paste of unittests. The last pair belonged to the same category was a model schema for a Django

[5] `http://doc.qt.io/qt-4.8/moc.html`

database.

3.7 Conclusions

The source control system upon which GitHub is built, Git, encourages forking projects and independent development of those forks. GitHub provides an easy interface for forking a project, and then for merging code changes back to the original projects. This is a popular feature: the metadata available from GHTorrent shows an average of 1 fork per project. However, there is a lot more duplication of code that happens in GitHub that does not go through the fork mechanism, and, instead, goes in via copy and paste of files and even entire libraries.

We presented an exhaustive investigation of code cloning in GitHub for four of the most popular object-oriented languages: Java, C++, Python and JavaScript. The amount of file-level duplication is staggering in the four language ecosystems, with the extreme case of JavaScript, where only 6% of the files are original, and the rest are copies of those. The Java ecosystem has the least amount of duplication. These results stand even when ignoring very small files. When delving deeper into the data we observed the presence of files from popular libraries that were copy-included in a large number projects. We also detected cases of reappropriation of entire projects, where developers take over a project without changes. There seemed to be several reasons for this, from abandoned projects , to slightly abusive uses of GitHub in educational contexts. Finally, we studied the JavaScript ecosystem, which turns out to be dominated by Node libraries that are committed to the applications' repositories.

This study has some important consequences. First, it would seem that GitHub, itself, might be able to compress its corpus to a fraction of what it is. Second, more and more research is being done using large collections of open source projects readily available from GitHub.

Code duplication can severely skew the conclusions of those studies. The assumption of diversity of projects in those datasets may be compromised. DéjàVu can help researchers and developers navigate through code cloning in GitHub, and avoid it when necessary.

Chapter 4

Method-level Duplication Analysis between Stack Overflow and GitHub

The material in this chapter is from the following paper, and is included here with permission from ACM.

D. Yang, P. Martins, V. Saini, and C.V. Lopes. Stack Overflow in Github: Any Snippets There? In proceedings of the 14th International Conference on Mining Software Repositories (MSR), May 2017.

4.1 Introduction

The popularity and relevance of the Question and Answer site Stack Overflow (SO) is well known within the programming community. As a measure of its populatiry, SO received more than half a billion views on the first 30 days of 2017 alone[1]. Another very popular site is

[1] `https://www.quantcast.com/stackoverflow.com` [Accessed January, 2017]

Github (GH), a project repository that ranked 14th on Forbes Cloud 100 in 2016[2] Although both sites are equally relevant for the programming community, they are so in different contexts. SO is a Q&A website with a strong community-based support, responsible for providing answers for virtually any type of programming problems and helping any type of user, from casual SHELL users to expert system administrators. GH also has a strong social component, but it is more focused on the storage and maintenance of software artifacts, providing version controlling features, bug management, control over the coder-base and contributors of projects, and so on.

Both platforms are part of a larger system of globalized software production. The same users that rely on the hosting and management characteristics of GH often have difficulties and need help on the implementation of their computer programs, seek support on SO for their specific problems, or hints of solutions from ones with a degree of similarity, and return to GH to apply the knowledge acquired. Empirically, however, there is little evidence of the actual impact that these two systems have on each other, or of the kind of information that goes from one platform to the other. Analyzing this relation is the focus of this work.

In isolation, SO has been the subject of various research studies. One example is the use of topic modeling on SO questions to categorize discussions [19, 21, 144], another is the use SO statistics to analyze use behavior and activity [36, 139]. Recent work has paid special attention to code snippets. Wong *et al.* [147] and Ponzanelli *et al.* [102] both mine SO for code snippets that are clones to a snippet in the client system. Yang *et al.* [149] provide a usability study of code snippets of four popular programming languages.

There are already some studies that investigate some relations between SO and GH. Vasilescu *et al.* [137] investigated the interplay between asking and answering questions on SO and committing changes to GH repositories. They answered the question of whether participation

[2] The Forbes Cloud 100 recognizes the top 100 private cloud companies in the world (`http://www.forbes.com/cloud100`).

in SO relates to the productivity of GH developers. From this work, we know that GH and SO overlap in a knowledge-sharing ecosystem: GH developers can ask for help on SO to solve their own technical challenges; they can also engage in SO to satisfy a demand for knowledge of others, perhaps less experienced than themselves. Moreover, we see this overlapping of knowledge also indicating another kind of overlapping: pieces of code. GH programmers can copy-paste SO code snippets to solve their particular problems; they can also use their existing code in GH repository to answer SO questions.

An *et al.* [18] conducted a case study with 399 Android apps, to investigate whether developers respect license terms when reusing code from SO posts (and the other way around). They found 232 code snippets in 62 Android apps that were potentially reused from SO, and 1,226 SO posts containing code examples that are clones of code released in 68 Android apps, suggesting that developers may have copied the code of these apps to answer SO questions.

In this study, our goal is to investigate and understand how much the snippets obtained from SO are used in GH projects. We opertionalize this problem as pieces of source code that exist in both sides, and we search for cloning and repetition as a measure of equal information presented in both places.

How much of the knowledge base, represented as source code, is shared between SO and GH? If SO and GH have overlapping source code, is this copy literal or does it suffer adaptations? And are these adaptations, if they exist, specializations required by the idiosyncrasies of the target or by the idiosyncrasies or the programmer, or both?

To answer these questions we perform intra and inter code duplication analysis on GH and SO. We uncover and document code duplicates in 909k Python projects from GH, which contain 292M function definitions in GH and 1.9M snippets in SO. Our choice of language is driven by popularity and by existing work by Yang *et al.* [149], which shows Python snippets in SO having one of the highest usability rates among the popular languages.

The rest of the paper is organized as follows. Section 4.3 details the methodology we applied to find code duplicates. Section 7.4 describes the datasets we used. Quantitative findings are presented in Section 4.5 and qualitative analysis in Section 4.6. Related work is present in Section 4.2. Section 6.7 concludes the paper.

4.2 Related Work

This paper involves different aspects of study, first, it focuses on the code itself of SO; second, it discovers the relationship between SO and GH; third, it investigates the large-scale code duplication detection in block-level, which includes uniqueness of source code. The related work come with these angles.

Wong et al. [147] devised a tool that automatically generates comments for software projects by searching for accompanying comments to SO code that are similar to the project code. They did so by relying on clone detection. This work is very similar to Ponzanelli et al. [101] [102] [103] in terms of the approach adopted. Both mine for SO code snippets that are clones to a snippet in the client system, but Ponzanelli et al.'s goal was to integrate SO into an Integrated Development Environment (IDE) and seamlessly obtain code prompts from SO when coding. In another work from Ponzanelli et al., they presented an Eclipse plugin, Seahawk, that also integrates SO within the IDE. It can add support to code by linking programming tools with SO search results.

There are two studies about assessing the usablity of code in SO. Nasehi et al. [89] engaged in finding the characteristics of a good example. They adopted a holistic approach and analyzed the characteristics of high voted answers and low voted answers. They enlisted traits by analyzing both the code and the contextual information: the number of code blocks used, the conciseness of the code, the presence of links to other resources, the presence of

alternate solutions, code comments, etc.

Yang [149] assessed the usability of SO snippet with a different criteria. They define usablity based on the standard steps of parsing, compiling and running the source code, which indicates that the effort that would be required to use the snippet as-is. A total of 3M code snippets are analyzed across four languages: C#, Java, JavaScript, and Python. Python and JavaScript proved to be the languages for which the most code snippets are usable. Conversely, Java and C# proved to be the languages with the lowest usability rate.

Vasilescu, et al. [137] investigated the interplay between SO activities and the development process, reflected by code changes committed to the largest social coding repository, GH. They found that active GH committers ask fewer questions and provide more answers than others, and active SO askers distribute their work in a less uniform way than developers that do not ask questions.

An *et al.* [18] aims to raise the awareness of the software engineering community about potential unethical code reuse activities taking place on Q&A websites like SO. They conducted a case study with 399 Android apps, to investigate whether developers respect license terms when reusing code from SO posts (and the other way around). From the 232 code snippets in 62 Android apps that were potentially reused from SO, and the 1,226 SO posts containing code examples that are clones of code released in 68 Android apps, they observed 1,279 cases of potential license violations (related to code posting to SO or code reuse from SO).

Some previous work has been done on code clone detection in block-level. Roy and Cordy [112] analyzed clones in twenty open source C, Java and C# systems, using the NiCad block-level clone detector. They found that on average 15% of the files in the C systems, 46% of the files in the Java systems and 29% of files in the C# systems are associated with exact (block-level) clones. Heinemann et al.[58] computed type-2 block-level clones between selected 22 commonly reused Java frameworks (e.g. Eclipse and Apache) and 20 open source

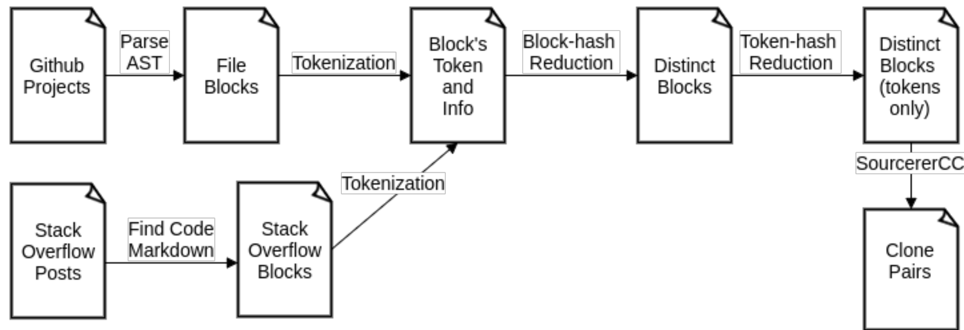


Figure 4.1: Pipeline for file analysis.

Java projects. They didn't find any clones for 11 of the 20 study objects. For 5 projeccts, they found cloning to be below 1% and for the remaining 4 projects, they found in the range of 7% to 10% cloning.

Gabel *et al.* [51] presented the results of the first study of *uniqueness of source code*. They gave *uniqueness* of a unit of source code a precise measure: synctactic redundancy. They wanted to figure out at what levels of granularity is software unique, and at a given level of granularity, how unique is software. We compute syntactic redundancy for 30 assorted SourceForge projects and 6,000 other projects. The results revealed a general lack of uniqueness in software at levels of granularity equivalent to approximately one to seven lines of source code. This phenomenon appears to be pervasive, crossing both project and programming language boundaries.

Hindle *et al.* [60] pointed out like natural language, software is also likely to be repetitive and predictable. Using n-gram model, they provided empirical evidence to support that code can be usefully modeled by statistical language models and such models can be leveraged to support software engineers. They showed that code is also very repetitive, and in fact even more so than natural languages.

4.3 Methodology

In this section, we describe the pipeline followed for analyzing block-level duplication inter and intra GH and SO.

Figure 7.2 contains the main steps in the analysis process. The pipeline starts by extracting blocks from GH projects and SO posts. Blocks from both origins are then scanned to obtain tokens and other relevant information (a process we call tokenization from now on).

In this analysis process, we provide three levels of similarity: a hash on the block, a hash on tokens, and an 80% token similarity. These capture the case where entire blocks are copied as-is, smaller changes are made in spacing or comments, and more meaningful edits are applied to the code.

Moreover, all the similarity analyses are done for intra-GH, intra-SO, and inter GH and SO. The following of this section will discuss each of the step in the pipeline in detail.

4.3.1 Method Extraction

For GH projects, our concept of block is that of a function definition. We extract the following two kinds of function definitions: (1) function defined inside of a class; (2) function defined outside of a class; For nested functions, we only consider the outtermost function. Consider the following example:

```
1 class Foo:
2   def func1(a, b, c):
3     return a + b
4
5 def func2(a, b, c):
6   if a>b:
7     return c
8   return 0
```

```
9
10 def func3(a):
11   def func4(b):
12     return b*2
13   return func4(3)
```

Listing 4.1: Github blocks

From the example above, we extract three functions: `func1`, `func2`, and `func3`. `func4` was nested inside of an already existing block, `func3`, and is therefore ignored.

The AST also exposes the starting and ending line numbers for its constituents, information we use to define blocks and contextualize them. Note that this is only possible in settings where a block resides within a file, such as GH; for SO the line numbers are useless.

In SO, both questions and answers are considered Posts, for which a unique id is associated. Posts are distinguished by a `PostTypeId` indicating if it is a question `PostTypeId=1` or an answer `PostTypeId=2`. The link between answers and their original questions is preserved. Only Question posts have tags marking the related languages and topics of the post, therefore all the pieces of code we process come from, or are related to, a Question whose tags contain 'python'. For all posts for Python, we used the markdown `<code>...</code>` to extract code snippets from Posts.

4.3.2 Tokenization

Tokenization is the process of transforming a file into a "bag of words". Tokenization involves removing comments, spaces, tabs and other special characters, identifying each individual word (token), and counting their frequency.

Consider the following Python block below:

```
1 def func1(a, b, c): # example block
2  if a>b: # condition
3    return c
4  else:
5    return 0
```

Listing 4.2: Github block tokenization

During tokenization, tokens in the block are identified and their occurrences are counted. The result after tokenizing the block in 4.2 is:

```
[(def, 1), (func1, 1), (a, 2), (b, 2), (c, 2),
(if, 1), (return, 2), (else, 1), (0, 1)]
```

where the token `def` and `func1` appear once, the tokens `a`, `b` and `c` appear twice, and so on.

During tokenization we also capture facts about blocks, specifically: (1) block hash: the MD5 hash of the entire string that composes the block; (2) token hash: the MD5 hash of the string that constitutes the tokenized block; (3) number of lines (4) number of lines of code: LOC (no blanks); (5) number of lines of source code: SLOC (no comments); (7) number of tokens; (8) number of unique tokens. For GH blocks, also (9) starting line; (10) ending line.[3]

4.3.3 Three Levels of Similarity

Two types of code clones are calculated simply based on hash values originated from two sources: the blocks themselves, and their tokenized forms.

The first type of clones, calculated by the hash values of their absolute composition of blocks (including spaces, all the characters, comments and so on) are called block-hash clones. When two blocks are block-hash equal, it means they are an exact copy of each other.

[3]There is some possibility that hash collisions will provide the same hashes for different blocks. Through relevant in the fields of cryptography and cryptosecurity, this is so unlikely we simply chose to ignore this possibility.

The second type of clones are calculated using their tokenized forms. These clones, called token-hash clones, differ from block-hash clones because they focus on the source code constituents of the blocks. Note that block-hash clones provide a very precise relation between two blocks, but has the consequence of being extremely sensible to small, irrelevant variations between blocks since any minimal difference of spaces, tabs, indentation or comments for example will flag two blocks as not clones. Therefore, we use tokenization to eliminate these small idiosyncrasies between two blocks that are irrelevant from a semantic perspective.

The first two levels of similarity are obtained by hash equality, being it at the block level or after its tokenization. These two levels do not reveal partial cloning, which in practice means certain scenarios where two blocks are cloned are not detected. Examples include familiar behaviors of literal copy-paste of a block, followed by a small specialization of a variable, or addition of tracing and debugging, actions through which intruders are inserted into the source code but their impact is so small that the blocks are still clones. This kind of problem if called near-miss clones in the area of code cloning.

To cover these scenarios, we use the tool SourcererCC [116], which has the capability of detecting relative similarities of two pieces of source code given a certain threshold. SourcererCC is a token-based clone detector, it can detect three types of clones. It also exploits an index to achieve scalability to large repositories using a standard workstation.

By evaluating the scalability, execution time, recall and precision of SourcererCC, and comparing it to four publicly available and state-of-the-art tools, SourcererCC has been shown to have both high recall and precision, and is able to scale to a large repository using a standard workstation. All of the above make SourcererCC a good candidate for this study. We used the default settings of SourcererCC, i.e., each clone pair has more than 80% of similarity.

Table 4.1: Github Dataset

# projects (total)	2,340,845
# projects (non-fork)	1,096,246
# projects (downloaded)	1,096,246
# projects (analyzed)	909,288
# files (analyzed)	31,609,117
# parsable files (analyzed)	30,986,363
# parsable blocks (analyzed)	290,742,628

4.4 Dataset

We downloaded the Github (GH) Python projects by using the metadata provided by GHTorrent [57**?** **?**]. GHTorrent is a scalable, offline mirror of data offered through the Github REST API, available to the research community as a service. It provides access to all the meta-data from GitHub, such as number of stars or commiters, main languages, time points relevant to the projects and so on.

For this work, we downloaded 909k Python non-fork repositories based on the GHTorrent's metadata available on November 2016. Filtering non-fork projects is an important constrain because through this mechanism information is necessarily cloned (direct replication is in the nature of forking a project) and therefore would skew the results.

Table 4.1 shows information regarding the entire corpus of Python projects that were used in this study. The gap between the projects that were downloaded and analyzed represents residual problems on accessing the downloaded information (typically corrupt zip archives, but also data on GHTorrent that was not up-to-date). The gap between analyzed and parsed files represents residual problems on parsing (for some reasons, Python's AST module [**?**] could not process them); only the latter, the parsed files, contribute to this study.

Figure 4.2 provides information regarding basic properties of the corpus of Python projects (note the first histogram is the only one demonstrating a 'per-project' property, the others

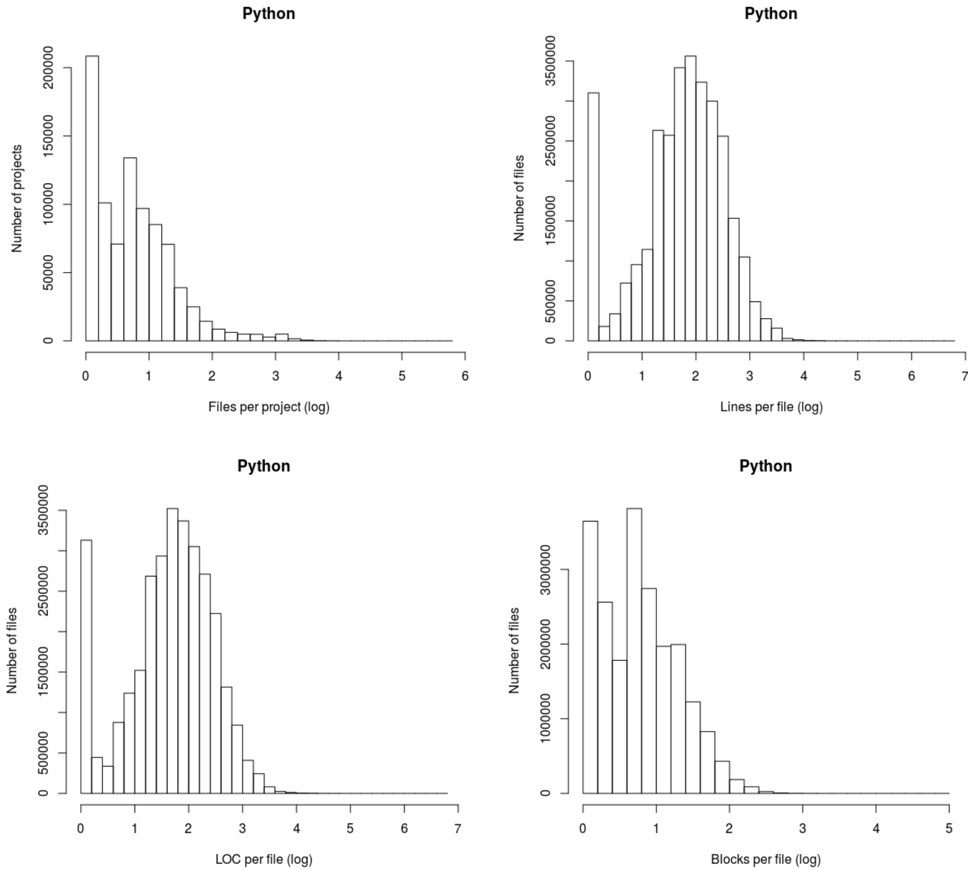


Figure 4.2: Python GitHub projects. LOC means Lines Of source Code, and is calculated after removing empty lines.

provide file's properties; and that the scale is logarithmic).

Stack Overflow (SO) has two sources of information (two type of Posts, from now on), typical of community-based online Q&A websites: one is the Question, and the other the Answers. All snippets were extracted from the dump available at the Stack Exchange data dump site [**?**].

A final note: we removed single-line Python snippets because these contain so little information that they are hardly representative. They typically exist in the context of larger snippets for which the users provide small comments, making them decontextualized in isolation.

Table 4.2 shows the total number of posts (questions and answers), number of Python blocks,

Table 4.2: Stack Overflow Dataset

# posts (total)	33,566,855
# posts (Python)	5,358,645
# blocks (Multiline)	1,954,025

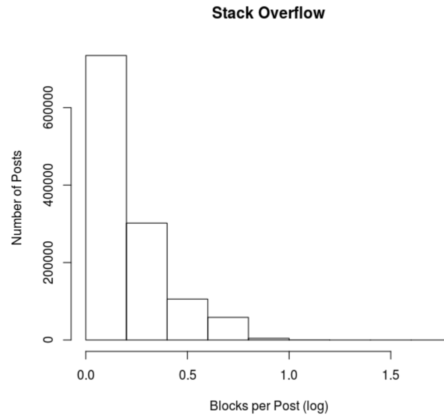


Figure 4.3: Blocks per Post.

and the number of multiple-line Python blocks on SO. Figure 4.3 shows the number of blocks per post.

Figure 4.4 represents a comparison between blocks originated from SO and GH. On top, we can see the distribution of the number of lines of source code (total lines minus empty lines), and in the bottom we can see the distribution of unique tokens. It is interesting to observe a high degree of similarity between blocks from the two origins on the two distributions. Understanding whether this similarity is a coincidence, or the object of transport of information from one source to the other will be the object of the research presented in next Sections.

4.5 Quantitative Analysis

In this section of we provide the values we found for code similarity between GH and SO.

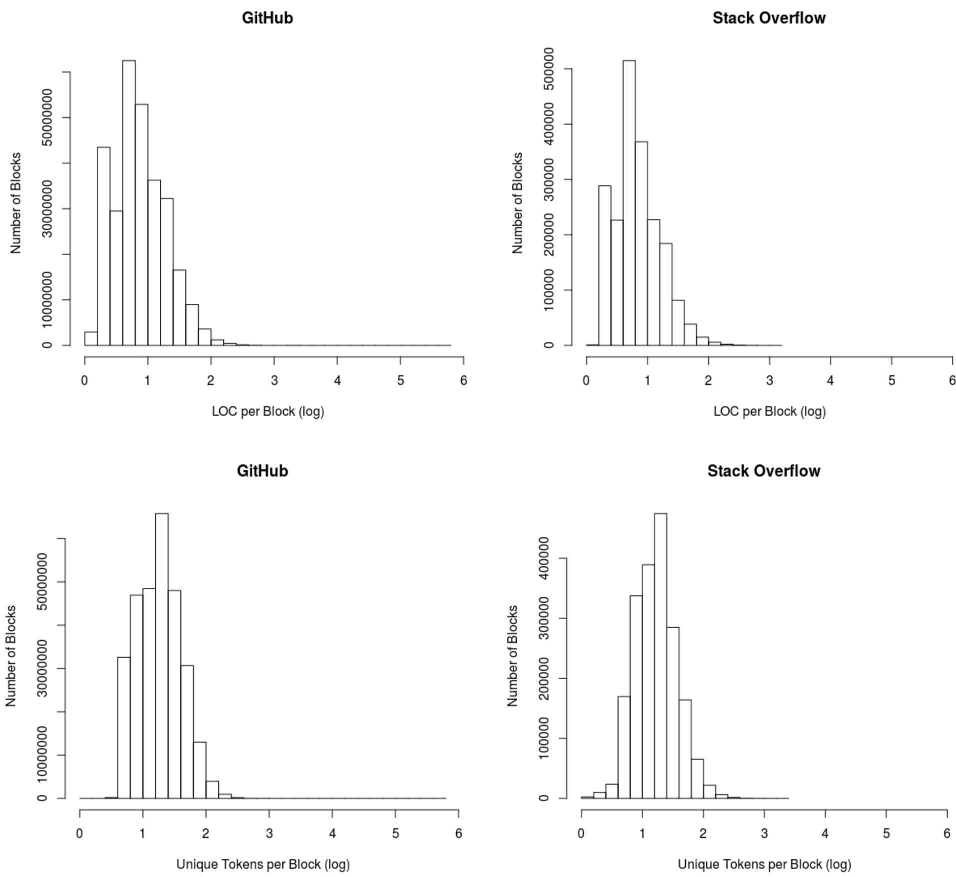


Figure 4.4: Per Block distributions of LOC (top) and unique tokens (bottom), in GitHub and Stack Overflow

Table 4.3: Block-hash similarity

	GH	SO
Total blocks	290,742,628	1,954,025
Distinct block-hashes	40,098,522	1,929,411
Common distinct block-hashes	1,566	1,566
Common blocks	60,962	2,091

We provide three types of analysis using, first, hash values on the blocks (block-hash), second, token hash on the blocks' source code (token-hash) and third, partial clones using SourcererCC. This provides different degrees of similarity for blocks: on the first we compare for perfect equality, on the second we filter glueing syntactic elements (spaces, tabs, terminal symbols, etc.), and on the third we allow some divergence.

Despite focusing this work on similarities between SO and GH, we always provide an individual analysis of each dataset. We do so to contextualize correlations between them from the perspective of each one individually.

4.5.1 Block-hash Similarity

The results for block-level hashing can be seen in Table 4.3. For hash analysis, we start by reducing the total group of blocks to a distinct set of block-level hashes. This set, shown on the second row of the table, represents the number of distinct pieces of code on the datasets. For GH, out of the 290M blocks there are only 40M distinct hashes, meaning that block-level code duplication is intense: 86% of blocks have the same exact code as the other 14%. This large amount of code duplication in open source project repositories has been observed before. For SO, the numbers are considerably smaller, with an almost absence of block duplication; only 1.3% of the blocks have the same code as the other 98.7%.

Next, we make the intersection of the distinct hashes in both datasets, obtaining the common distinct hashes between GH and SO. That number is shown in the third row: 1,566. This is

Table 4.4: Token-hash similarity

	GH	SO
Total # blocks	290,742,628	1,954,025
Distinct token-hashes	35,894,897	1,890,565
Common distinct token-hashes	9,044	9,044
Common blocks	3,839,019	13,747

a very small percentage of the distinct hashes in both datasets.

Finally, in row four we count all the blocks whose hashes belong to the common hashes. These are the blocks of code that exist in their exact form, including formatting and whitespace, in both GH and SO. The percentages are very small, less than 1% in both cases.

4.5.2 Token-hash Similarity

The results for block-hash analysis are presented in Table 4.4. Not surprisingly, when formatting and whitespace are ignored, the code duplication in each dataset increases slightly, i.e. the number of distinct token hashes is smaller than the number of distinct block hashes (compare second rows of Tables 4.3 and 4.4).

For the same reason, the common distinct token hashes between GH and SO is considerably larger than the common distinct block hashes (compare third rows of Tables 4.3 and 4.4). But the percentage of distinct hashes that are common to both datasets is still very small.

Interestingly, the number of blocks in GH whose token hashes are in the common set is above 1% (see row four). While small, it is remarkable that so many Python functions in GH projects, almost 4M, have the exact same tokens as snippets of Python code found in SO.

Table 4.5: SCC Similarity

	GH	SO
Distinct token hashes	35,894,897	1,890,565
SCC-dup	13,363,759	297,554
Common	405,393	35,098

4.5.3 SCC Similarity

The analysis in this subsection is slighly different than in the previous two: we narrow the analysis only to the universe of blocks that have distinct token hashes, those counted in the second row of Table 4.4. The rationale is that two files with the same token-hashes will be detected as clones by SCC, and therefore it suffices to process only one representative of each group of blocks with the same token hash.

The results are presented in Table 4.5. The second row, SCC-dup, shows the number of blocks in each dataset that have at least one similar block in the same dataset – only within the universe of distinct token hashes. The amount of near-duplication is considerably high in GH (roughly, 37%), but less in SO (roughly 16%).

The third row shows the number of blocks that are similar between datasets – again, only within the universe of distinct token hashes. More than 405k (1.1%) of the blocks in GH are similar to blocks in SO, and 2% of blocks in SO are similar to blocks in GH. This means that 35,098 distinct blocks found in SO can be found in very similar form in GH. The number is considerably larger than the common distinct token-hashes in Table 4.4.

4.6 Qualitative Analysis

To understand the nature of blocks that can be found in both SO and GH, we made a qualitative analysis on the duplicated blocks. This analysis was made in two steps. We first

looked at subsets of all of them, looking for patterns. One strong pattern emerged: the majority of blocks that are duplicated – both within datasets as well as between them – are very small, typically a couple of lines of code. These also tend to be non-descriptive, with very generic code (e.g. trivial `__init__` methods). Having observed this, we then moved to a second stage of analysis, where we looked only at larger functions. The number of these blocks is much smaller, but they are more interesting. This section describes our qualitative analysis.

4.6.1 Step 1: Duplicated Blocks

We looked the top 10 most duplicated code blocks based on their block-hash, token-hash, and SourcererCC reported clones. We did this analyses for intra-GH and intra-SO block clones. Further, to understand the kind of code blocks which are common across GitHub and SO, we looked at the 10 code blocks which are present in both GitHub and SO, and are duplicated the most in GitHub and similarly the top 10 code blocks which are duplicated the most in SO. The duplicated code blocks were selected based on block-hash, token-hash, and SourcererCC reported clones.

4.6.1.1 Block-Hash Duplicates

Intra-GH: All of the top 10 duplicated code-blocks had 2 lines of code. Four of these methods can be traced back to `cp037.py` file, located at `https://github.com/python-git/python/blob/master/Lib/encodings/cp037.py`. The file gets generated from `'MAPPINGS/VENDORS/MICSFT/EBCDIC/CP037.TXT'` with `gencodec.py` as mentioned in the file level comment inside the file. There are many such files, each for a different encoding `cp1253`, `cp1026`, `cp1140`, and so on. Further, we found many

instances where these files are present in other GitHub projects. Each of these files contains the generic methods to encode and decode the input string, as shown in the Listing 4.3 below.

```
1 class Codec(codecs.Codec):
2
3 def encode(self,input,errors='strict'):
4 return codecs.charmap_encode(input,errors,encoding_table)
5
6 def decode(self,input,errors='strict'):
7 return codecs.charmap_decode(input,errors,decoding_table)
```

Listing 4.3: Most duplicated code block based on Block-Hash

Other most duplicated code blocks are private methods `__iter__`, `__enter__`, `__ne__`, and `__init__`, with just one statement, as shown in Listing 4.4.

```
1 def __iter__(self):
2 return self
```

Listing 4.4: Example of one highly duplicated private method

Intra-SO: Like GH, the top 10 most duplicated code blocks on SO have two lines of code. Listing 4.5 shows the most duplicated ones. The first code block (top), shows Python idiom for the main entry point in a module. The second code block from the top, prints out all directories which are on the python's path. This is usually done to fix issues related to the import of third party libraries. The bottom two blocks, also very common on SO, do not have any Python specific code, and are used to present an example output of some Python code.

```
1 if __name__ == '__main__':
2 main()
```

Listing 4.5: Most duplicated code blocks on SO based on Block-Hash

```
1 import sys
2 print sys.path
```

```
1 True
2 False
```

```
1 1
2 2
```

Most duplicated blocks in GH that are also present in SO: Listing 4.6 shows two of the most duplicated blocks in GH that are also present in SO. We found the code block for `session()` on SO, where it is mentioned that this code blocks was copied from the sessions module under the requests library. We found that a lot of projects on GH use this library, where they copy the entire source code. `__iter__` is a very common private function used to make a class iterable, and hence this code block is also duplicated a lot. On SO we found a post where this code block was used as an example to demonstrate how to make a class iterable.

```
1 def session():
2 """Returns a :class:‘Session‘ for context-management."""
3
4 return Session()
```

Listing 4.6: Most duplicated code blocks on GH, which are also present in SO based on Block-Hash

```
1 def __iter__(self):
2 return self
```

We also found some code blocks which are related to *Django*, a python web framework. These code blocks are not intentionally copied, and become a part of the projects that are using *Django*. Listing 4.7 shows an example code block. On inspection we found that this code block was copied to SO from GH, to show the code in Django which is responsible for creating an anonymous user.

```
def get_user(request):
from django.contrib.auth.models import AnonymousUser
try:
user_id = request.session[SESSION_KEY]
backend_path = request.session[BACKEND_SESSION_KEY]
backend = load_backend(backend_path)
user = backend.get_user(user_id) or AnonymousUser()
except KeyError:
user = AnonymousUser()
return user
```

Listing 4.7: Most duplicated code blocks on GH, which are also present in SO based on Block-Hash

An observation common to most of these code blocks is that these blocks get duplicated in GH not because developers are interested in a particular code block, but because they are interested in the entire module like modules from *requests* library, or because they are using a framework which adds the source files into the projects. On SO, users are more interested in explaining a particular behavior or seeking some explanation about code blocks. We observed such scenarios where users have used a code block from GH and have also pasted the link of the source file in GH.

Most duplicated blocks in SO that are also in GH: Interestingly, 8 out of the top 10 code blocks come from `itertools` `https://docs.python.org/2/library/itertools.html#itertool-functions`. To understand the origin of these code blocks on SO, we looked at the two most duplicated ones, shown in Listing 4.8. On SO, we found 28 instances of the block on top, `any()` and 24 instance of the one in bottom, `grouper()`. On SO, we looked at 5 random instances of `any()` and found that this code block was copied from `https://docs.python.org/2/library/functions.html#any` and not from GH. We could link the origin based on the comments written on the SO posts. On GH we found some projects which have `any.py` module implementing the exact code block. We also found modules on GH which

implement code blocks similar to `any` like `all`, `enumerate`. Some of these modules come from projects where it was quite evident that the user has copied code into their project. For example, a project where a duplicate of `any()` function found, mentions in its `README.md`: *I want to collect something that I think it's interesting. Maybe some code snippet I think it's excellent cool.*

We made a similar observation when we looked at the origin of `grouper()`. In many instances on SO, the code block was copied from the python docs. On GH, we found one instance of this code block at `https://github.com/hbradlow/dynamic_path/blob/master/path/utils.py`. We also observed a comment in the same file with a url to a SO post. On further inspection we found that the most of the code in the module was copied from the SO post.

```
1 def any(iterable):
2 for element in iterable:
3 if element:
4 return True
5 return False
```

Listing 4.8: Most duplicated code blocks on SO, which are also present in GH based on Block-Hash

```
1 def grouper(n, iterable, fillvalue=None):
2 "grouper(3, 'ABCDEFG', 'x') --> ABC DEF Gxx"
3 args = [iter(iterable)] * n
4 return izip_longest(fillvalue=fillvalue, *args)
```

4.6.1.2 Token-Hash Duplicates

Intra-GH: To analyze Token-Hash duplicates, we followed a process similar to what we used for analyzing Block-Hash duplicates. The observations are very similar to those made

in the Block-Hash duplicates section. Most duplicated code blocks are `encode`, `decode`, `__iter__`, `__enter__`, `__ne__`, and `__init__`, as shown in the Listings 4.3 and 4.4.

Intra-SO: We found that the code blocks that resulted into 0 tokens were reported as the most duplicated code blocks. These are the blocks where all statements are commented, for example consider a code block shown in Listing 4.9. This blocks will generate 0 tokens as comments are ignored during tokenization. The token hash of all such blocks will be computed on an empty string, resulting into same token-hash.

```
1 #define private public
2 #include <module>
```

Listing 4.9: Example of 0 token code block

Listing 4.10 shows examples of duplicated code blocks on SO. The first code block (top), shows an example of how to instantiate three different list objects. The SO post for this code block is full of similar examples.

```
1 a = []
2 b = []
3 c = []
```

Listing 4.10: Most duplicated code blocks on SO, based on Token-Hash

```
1 1 2 3
2 4 5 6
3 7 8 9
```

The second code block from the top, shows a representation of a two dimensional list. SO, has many such blocks, where users have used this representation to explain the desired output of their code. 5 out of top 10 duplicated blocks on SO are about lists of numbers. We also observed many code blocks similar to shown in Listing 4.5.

Most duplicated blocks in GH that are also present in SO: The results found are

mostly shared code represents simple two liners and are very trivial, such as `__ne__` or `__str__`.

Most duplicated blocks in SO that are also in GH: Similarly to the results on the opposite direction, the blocks we found were of a small dimension and were characterized by trivial information.

Token hash vs Block hash We ignored the output explanation blocks, and dig into the reason for the code block pairs being caught as duplicates for token-hash level instead of block-hash level. We found that most of the pairs were only different in spaces. Some are token-hash duplicates because of the difference in the syntax of Python 2 and Python 3, for example in the `print` function. A few are token-hash duplicates because some parameter or variable is set to be am empty list, which results in differences in special characters, for example in the pair in List 4.11.

```
def __init__(self, connection):
self.connection = connection
```

Listing 4.11: Example for difference in token-hash duplicates

```
def __init__(self, connection=[]):
self.connection = connection
```

4.6.1.3 SourcererCC Duplicates

The qualitative analysis of block-hash duplicates and token-hash duplicates hints at exact copy-paste inside and between GH and SO. However, from the SCC results in the quantitative analysis, we learned that there are many cases where programmers make adaptations to the codes during copy-pasting. Therefore, here we want to see how people change their code when inside and between GH and SO.

Intra-GH: All of the top 10 most duplicated blocks we found inside GH are from the same file, located at `https://github.com/lufo816/WeiXinCookbook/blob/master/urlHandler.py`. This file has 80,452 clones similar to the block on Listing 4.12.

```
1 def GET(self):
2 return render.caipu1()
```

Listing 4.12: Most duplicated code block intra GH from SCC

The only difference on these blocks is the number in the function, ranging from 1 to 80,452. SCC will take every block as a clone for all other blocks in this same file, so they become the most duplicated blocks.

Intra-SO: Here, we found that 5 of the 10 examples are python error message of `ImportError`, similar to the one on Listing 4.13, with the difference in module name or line number. There were 4 blocks that are standard settings for *Django* and the remaining one is a list of numbers representing an output, similar to what we have seen above.

```
1 Traceback (most recent call last):
2 File "<stdin>", line 1, in <module>
3 ImportError: No module named MySQLdb
```

Listing 4.13: A common error message

Most duplicated blocks in GH that are also present in SO: Overall, 7 out of 10 of the blocks we analyzed can be traced to modules from libraries like `requests` and `pip`, such as the examples of Listing 4.14.

```
1 def __ne__(self, other):
2 return not self.__eq__(other)
```

Listing 4.14: Most duplicated code blocks on SO, which are also present in GH based on the results from SCC

```
def __init__(self, username, password):
self.username = username
self.password = password
```

Most duplicated blocks in SO that are also present in GH: For the top 10 most duplicated blocks in SO that are also present in GH, there are actually only three kinds of blocks as shown in 4.15. The first is a standard initial function for a class; the second is a standard function definition with parameters, the third is a function that raise `NotImplementedError`. The first group contains 5 pairs, the second contains 4 pairs, and the third only has 1 pair.

```
def __init__(self):
self.locList = []
```

Listing 4.15: Most duplicated code blocks on SO, which are also present in GH based on SCC

```
def some_function(*args, **kwargs):
pass
```

```
def number_of_edges(self):
raise(NotImplementedError)
```

When observing the SCC clone for each block, we found that for in group 1, all pairs contain the tokens `def`, `raise`, and `self`, and the only difference is the adaptation to specific variables. For group 2, all clone pairs contain tokens `def`, `*args`, `**kwargs`, `pass`, the only difference is the function name. For the block in group 3, its clone pair have the same tokens `def`, `self`, `raise` and `NotImplementedError` and the only difference is the function name.

SCC vs Token hash From the observations above, we can see that the reason for these pairs being duplicates in SCC level instead of token-hash level is changes of function names, parameters, or variables.

4.6.2 Step 2: Large Blocks Present in GitHub and SO

To further understanding the correlation and copy-paste behavior between GH and SO, we set a threshold to the number of unique tokens in a block to get larger blocks for observation.

4.6.2.1 Block-Hash Duplicates

We set the threshold of unique tokens of each block to be equal or larger than 30 tokens in order to filter out meaningless small blocks. This filtering left us with 104 common block hashes between GH and SO, from the original 1,566 common block hashes. We sampled 10 block hashes out of these 104 and traced one sample pair of GH and SO blocks for each common block hash.

From the 10 pairs we got, 4 of the GH blocks explicitly stated in the comments that the code was borrowed from SO, and also gives the SO post link corresponding to the block. The SO post links were exactly the same as we paired for the GH block. This is a very clear evidence source code has been flowing from SO to GH.

In two pairs, GH and SO blocks are coming from the same third-party source. In another three pairs, the SO post stated that the code was copied from a third-party source, but there's no explicit clue of where the GH block comes from, although there was only one commit on the file and no changes before and after, which may indicate the code was copied from other sources too.

4.6.2.2 Token-Hash Duplicates

The number of unique tokens is set to be equal or larger than 35 for token hash duplicates. We have 915 common token hashes between GH and SO after the filtering. For large token-hash duplicates, we observed a clear case of copy-paste from GH to SO, where the author of the code on GH used his own code as an example to demonstrate aspects of Python's `func_code` attribute.

Another relevant example is where a closer inspection of the comments on SO pointed directly the original website from where these blocks were copied, which happens to be the now defunct Google code. There are two clear indications of the transfer of knowledge from one source to the other.

Then we furthered observed the reason for the pairs being caught as duplicates only by token hash instead of block hash. Although token hash will leaving out all the comments, spaces, special characters, nine out of the ten sampled pairs only different in spaces, and all contents are kept as-is, including comments; only one pair is different in missing one line of comments. It means that during the process of copy-pasting large blocks of code, either between GH and SO or from other sources, programmers tend to preserve everything instead of dropping or changing any of them. This may because on one hand, the large blocks are a complete implementation of some functionality, and plugging them as-is is sufficient for programmers' needs and no changes needed; on the other hand, copy-paste is also a process of learning, and the comments help the learner understand what the code is about, so there's no point of deleting them intentionally.

4.6.2.3 SourcererCC Duplicates

The number of unique tokens is set to be equal or larger than 35 for SCC duplicates. We have 4699 distinct token hashes in SO that can be found very similar form in GH.

Using SCC we found many cases where code blocks on SO were similar to that on GH. On observing the blocks manually, it was hard to find clues that point at the directional of information exchange. In some cases it was obvious that deliberate copy-paste has resulted into code duplication, but we cannot say for sure whether the code was copied from GH to SO, SO to GH or from a third party website to GH or SO.

SCC marked these pairs as 80% similarity in tokens. We observed that the differences between them came from variables, function identifiers, `if` conditions, or `class` definition. In other words, when copy-pasting codes, programmers will adjust the variables, switch function names or parameters, change, add, or delete `if` conditions, or add or delete `class` definition to match their particular needs.

4.7 Conclusion

Stack Overflow, a popular Q&A site, has become one of the major Internet hubs where programmers can find all sorts of information related to simple, but concrete programming problems. We wanted to find out the extent to which the code snippets in SO find their way to open source projects. For this study, we focused on programs written in Python. As datasets, we took the collection of 909k non-forked Python projects hosted in Github, as well as the SO dump provided by Stack Exchange. We extracted all the multi-line Python code snippets from SO, and we parsed all the Python projects, breaking them into functions. We then cross referenced the SO snippets with these functions, using three measures of similarity: exact match, match on the tokens and near-duplication as detected by a code clone detector tool.

Our quantitative analysis shows that exact duplication between SO and GH exists, but is rare, much less than 1%. Token-level duplication is more common, with almost 4M blocks in

GH being similar to SO snippets. In terms of percentage, this is still small. Near-duplication shows 405k distinct blocks (1.1%) in GH being similar to SO and 35k (2%) SO distinct blocks having near duplicates in GH. Although the percentages are not very large, the numbers are in thousands.

Upon careful qualitative analysis, we observed that the vast majority of these duplicates are very small, typically 2 lines of code and just a few tokens. Moreover, they tend to be non-descriptive, meaning that they are too generic to trace. Because they are generic and small, likely they didn't flow from SO to GH or vice versa. We then focused out attention to the fewer blocks that are not so small. For these, we found evidence that there is, indeed, flow from SO to GH, in some cases that flow being explicitly stated in comments. While there is a lot less of these, their number is still in the thousands.

The importance of this work is twofold. First, it gives empirical evidence of the phenomenon of copy-and-paste from SO, something that is widely accepted to be true, but hasn't been studied. Second, the non-trivial SO snippets that can be found in real code in GH could be used as the basis for novel search engines for program synthesis and repair that integrate with the rich natural language descriptions found in SO. We found that there are 5,718 large blocks with distinct hashes in SO that can be found very similar form in GH. These large distinct blocks can be made good use of in the future work.

Enriched by natural language contexts surrounding the code snippets, SO can help to retrieve code snippets by matches on the non-coding information. Moreover, it can potentialy be used as a knowledge base for tools that automatically combine snippets of code in order to obtain more complex behavior. The viability of using SO in program synthesis lies, first of all, on the existence of good snippets and evidence that they exist in real code, which is shown in this paper.

Chapter 5

Analysis of Adaptations from Stack Overflow to GitHub

The material in this chapter is part of the following paper, and is included here with permission from ACM.

T. Zhang, D. Yang, C. V. Lopes, M. Kim. Analyzing and Supporting Adaptation of Online Code Examples. In proceedings of the 41st International Conference on Software Engineering (ICSE), May 2019.

This paper investigate the common adaptation types and their frequencies between Stack Overflow snippets and their GitHub counterparts. The study involved close collaborations between the UCLA group and us. I contributed to the paper by first collecting all the data needed from Stack Overflow and GitHub, detecting all similar method pairs between Stack Overflow and GitHub, then manually labeling adaptation types for some sampled Stack Overflow snippets and their GitHub counterparts, and finally implementing part of the automated techniques for categorizing adaptation types. In this chapter, I only included the parts that I was involved in.

5.1 Introduction

Nowadays, a common way of quickly accomplishing programming tasks is to search and reuse code examples in online Q&A forums such as Stack Overflow (SO) [29, 52, 135]. A case study at Google shows that developers issue an average of twelve code search queries per weekday [114]. As of July 2018, Stack Overflow has accumulated 26M answers to 16M programming questions. Copying code examples from Stack Overflow is common [23] and adapting them to fit a target program is recognized as a top barrier when reusing code from Stack Overflow [148]. SO examples are created for illustration purposes, which can serve as a good starting point. However, these examples may be insufficient to be ported to a production environment, as previous studies find that SO examples may suffer from API usage violations [152], insecure coding practices [46], unchecked obsolete usage [156], and incomplete code fragments [131]. Hence, developers may have to manually adapt code examples when importing them into their own projects.

Our goal is to investigate the common adaptation types and their frequencies in online code examples, such as those found in Stack Overflow, which are used by a large number of software developers around the world. To study how they are adopted and adapted in real projects, we contrast them against similar code fragments in GitHub projects. The insights gained from this study could inform the design of tools for helping developers adapt code snippets they find in Q&A sites.

In broad strokes, the design and main results of our study are as follows. We link SO examples to GitHub counterparts using multiple complementary filters. First, we quality-control GitHub data by removing forked projects and selecting projects with at least five stars. Second, we perform clone detection [116] between 312K SO posts and 51K non-forked GitHub projects to ensure that SO examples are similar to GitHub counterparts. Third, we perform timestamp analysis to ensure that GitHub counterparts are created later than the

SO examples. Fourth, we look for explicit URL references from GitHub counterparts to SO examples by matching the post ID. As the result, we construct a comprehensive dataset of *variations* and *adaptations*.

When we use all four filters above, we find only 629 SO examples with GitHub counterparts. Recent studies find that very few developers explicitly attribute to the original SO post when reusing code from Stack Overflow [18, 23, 148].

Therefore, we use this resulting set of 629 SO examples as an *under-approximation* of SO code reuse and call it an *adaptations* dataset. If we apply only the first three filters above, we find 14,124 SO examples with GitHub counterparts that represent potential code reuse from SO to GitHub. While this set does not necessarily imply any causality or intentional code reuse, it still demonstrates the kinds of common variations between SO examples and their GitHub counterparts, which developers might want to consider during code reuse. Therefore, we consider this second dataset as an *over-approximation* of SO code reuse, and call it simply a *variations* dataset.

We randomly select 200 clone pairs from each dataset and manually examine the program differences between SO examples and their GitHub counterparts. Based on the manual inspection insights, we construct an adaptation taxonomy with 6 high-level categories and 24 specialized types. We then develop an automated adaptation analysis technique built on top of GumTree [45] to categorize syntactic program differences into different adaptation types. The precision and recall of this technique are 98% and 96% respectively. This technique allows us to quantify the extent of common adaptations and variations in each dataset. The analysis shows that both the adaptations and variations between SO examples and their GitHub counterparts are prevalent and non-trivial. It also highlights several adaptation types such as type conversion, handling potential exceptions, and adding `if` checks, which are frequently performed yet not automated by existing code integration techniques [40, 146].

In summary, this work makes the following contributions:

- It makes publicly available a comprehensive dataset of *adaptations* and *variations* between SO and GitHub.[1] The adaptation dataset includes 629 groups of GitHub counterparts with explicit references to SO posts, and the variation dataset includes 14,124 groups. These datasets are created with care using multiple complementary methods for quality control—clone detection, time stamp analysis, and explicit references.
- It puts forward an adaptation taxonomy of online code examples and an automated technique for classifying adaptations. This taxonomy is sufficiently different from other change type taxonomies from refactoring [49] and software evolution [48, 70], and it captures the particular kinds of adaptations done over online code examples.

The rest of the paper is organized as follows. Section 7.4 describes the data collection pipeline and compares the characteristics of the two datasets. Section 5.4 describes the adaptation taxonomy development and an automated adaptation analysis technique. Section 5.5 describes the quantitative analysis of adaptations and variations. Section 3.7 discusses threats to validity. Section 5.2 discusses related work, and Section 6.7 concludes the paper.

5.2 Related Work

Quality assessment of SO examples. Our work is inspired by previous studies that find SO examples are incomplete and inadequate [18, 42, 46, 123, 149, 152, 156]. Subramanian and Holmes find that the majority of SO snippets are free standing statements with no class or method headers [123]. Zhou et al. find that 86 of 200 accepted SO posts use deprecated APIs but only 3 of them are reported by other programmers [156]. Fischer et al. find that

[1] Our dataset and tool are available at `https://github.com/tianyi-zhang/ExampleStack-ICSE-Artifact`

29% of security-related code in SO is insecure and could potentially be copied to one million Android apps [46]. Zhang et al. contrast SO examples with API usage patterns mined from GitHub and detect potential API misuse in 31% of SO posts [152]. These findings motivate our investigation of adaptations and variations of SO examples.

Stack Overflow usage and attribution. Our work is motivated by the finding that developers often resort to online Q&A forums such as Stack Overflow [23, 29, 114, 148]. Despite the wide usage of SO, most developers are not aware of the SO licensing terms nor attribute to the code reused from SO [18, 23, 148]. Only 1.8% of GitHub repositories containing code from SO follow the licensing policy properly [23]. Almost one half developers admit copying code from SO without attribution and two thirds are not aware of the SO licensing implications. Based on these findings, we carefully construct a comprehensive dataset of reused code, including both explicitly attributed SO examples and potentially reused ones using clone detection, timestamp analysis, and URL references.

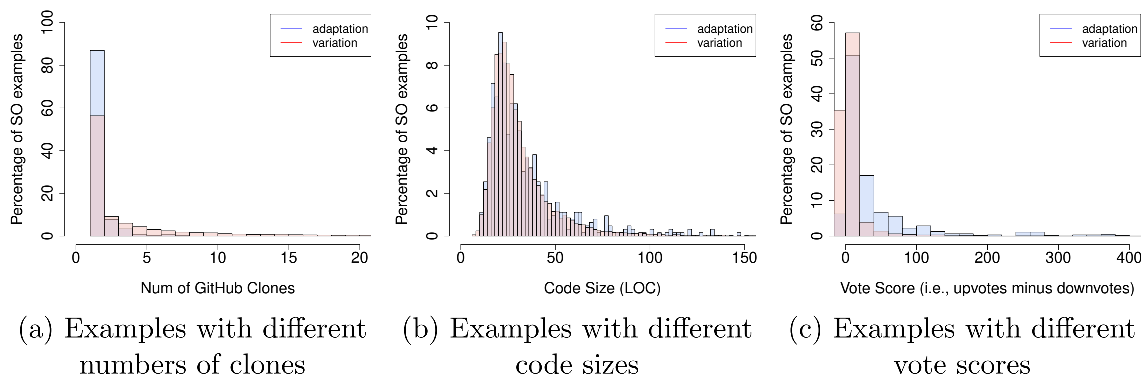
Origin analysis can also be applied to match SO snippets with GitHub files [54, 55, 133, 157]. ***SO snippet retrieval and code integration.*** Previous support for reusing code from SO mostly focuses on helping developers locate relevant posts or snippets from the IDE [20, 101, 102, 146]. For example, Prompter retrieves related SO discussions based on the program context in Eclipse. SnipMatch supports light-weight code integration by renaming variables in a SO snippet based on corresponding variables in a target program [146]. Code correspondence techniques [40, 63] match code elements (e.g., variables, methods) to decide which code to copy, rename, or delete during copying and pasting. Our work differs by focusing on analysis of common adaptations and variations of SO examples.

Change types and taxonomy. There is a large body of literature for source code changes during software evolution [43, 47, 69]. Fluri et al. present a fine-grained taxonomy of code changes such as changing the return type and renaming a field, based on differences in abstract syntax trees [48]. Kim et al. analyze changes on “micro patterns” [53] in Java using

software evolution data [70]. These studies investigate general change types in software evolution, while we quantify common adaptation and variation types using SO and GitHub code.

5.3 Dataset

This section describes the data collection pipeline. Due to the large portion of unattributed SO examples in GitHub [18, 23, 148], it is challenging to construct a complete set of reused code from SO to GitHub. To overcome this limitation, we apply four quality-control filters to *underapproximate* and *overapproximate* code examples reused from SO to GitHub, resulting in two complementary datasets.



(a) Examples with different numbers of clones (b) Examples with different code sizes (c) Examples with different vote scores

Figure 5.1: Comparison between SO examples in the adaptation dataset and the variation dataset

GitHub project selection and deduplication. Since GitHub has many toy projects that do not adequately reflect software engineering practices [65], we only consider GitHub projects that have at least five stars. To account for internal duplication in GitHub [75], we choose non-fork projects only and further remove duplicated GitHub files using the same file hashing method as in [75], since such file duplication may skew our analysis. As a result, we download 50,826 non-forked Java repositories with at least five stars from GitTorrent [57]. After deduplication, 5,825,727 distinct Java files remain.

Detecting GitHub candidates for SO snippets. From the SO dump taken in October 2016 [17], we extract 312,219 answer posts that have `java` or `android` tags and also contain code snippets in the `<code>` markdown. We consider code snippets in answer posts only, since snippets in question posts are rarely used as examples. Then we use a token-based clone detector, SourcererCC (SCC) [116] to find similar code between 5.8M distinct Java files and 312K SO posts. We choose SCC because it has high precision and recall and also scales to a large code corpus. Since SO snippets are often free-standing statements [123, 149], we parse and tokenize them using a customized Java parser [124]. Prior work finds that larger SO snippets have more meaningful clones in GitHub [150]. Hence, we choose to study SO examples with no less than 50 tokens, not including code comments, Java keywords, and delimiters. We set the similarity threshold to 70% since it yields the best precision and recall on multiple clone benchmarks [116]. We cannot set it to 100% since SCC will then only retain exact copies and exclude those adapted code. We run SCC on a server machine with 116 cores and 256G RAM. It takes 24 hours to complete, resulting in 21,207 SO methods that have one or more similar code fragments (i.e., clones) in GitHub.

Timestamp analysis. If the GitHub clone of a SO example is created before the SO post, we consider it unlikely to be reused from SO and remove it from our dataset. To identify the creation date of a GitHub clone, we write a script to retrieve the Git commit history of the file and match the clone snippet against each file revision. We use the timestamp of the earliest matched file revision as the creation time of a GitHub clone. As a result, 7,083 SO examples (33%) are excluded since all their GitHub clones are committed before the SO posts.

Scanning explicitly attributed SO examples. Despite the large portion of unattributed SO examples, it is certainly possible to scan GitHub clones for explicit references such as SO links in code comments to confirm whether a clone is copied from SO. If the SO link in a GitHub clone points to a question post instead of an answer post, we check whether the

corresponding SO example is from any of its answer posts by matching the post ID. We find 629 explicitly referenced SO examples.

Overapproximating and underapproximating reused code. We use the set of 629 explicitly attributed SO examples as an *underapproximation* of reused code from SO to GitHub, which we call an *adaptation* dataset. We consider the 14,124 SO examples after timestamp analysis as an *overapproximation* of potentially reused code, which we call a *variation* dataset. Figure 5.1 compares the characteristics of these two datasets of SO examples in terms of the number of GitHub clones, code size, and vote score (i.e., upvotes minus downvotes). Since developers do not often attribute SO code examples, explicitly referenced SO examples have a median of one GitHub clone only, while SO examples have a median of two clones in the variation dataset. Both sets of SO examples have similar length, 26 vs. 25 lines of code in median. However, SO examples from the adaptation dataset have significantly more upvotes than the variation dataset: 16 vs. 1 in median. In the following sections, we inspect, analyze, and quantify the adaptations and variations evidenced by both datasets.

5.4 Adaptation Type Analysis

5.4.1 Manual Inspection

To get insights into adaptations and variations of SO examples, we randomly sample SO examples and their GitHub counterparts from each dataset and inspect their program differences using GumTree [45]. Below, we use "adaptations" to refer both adaptations and variations for simplicity.

The first and the last authors jointly labeled these SO examples with adaptation descriptions and grouped the edits with similar descriptions to identify common adaptation types. We

Table 5.1: Common adaptation types, categorization, and implementation

Category	Adaptation Type	Rule
Code Hardening	Add a conditional	Insert(t_1, t_2, i) ∧ NodeType(t_1, IfStatement)
	Insert a final modifier	Insert(t_1, t_2, i) ∧ NodeType(t_1, Modifier) ∧ NodeValue(t_1, final)
	Handle a new exception type	Exception(e, GH) ∧ ¬Exception(e, SO)
	Clean up unmanaged resources (e.g. close a stream)	(LocalCall(m, GH) ∨ InstanceCall(m, GH)) ∧ ¬LocalCall(m, SO) ∧ ¬InstanceCall(m, SO) ∧ isCleanMethod(m)
Resolve Compilation Errors	Declare an undeclared variable	Insert(t_1, t_2, i) ∧ NodeType(t_1, VariableDeclaration) ∧ NodeValue(t_1, v) ∧ Use(v, SO) ∧ ¬Def(v, SO)
	Specify a target of method invocation	InstanceCall(m, GH) ∧ LocalCall(m, SO)
	Remove undeclared variables or local method calls	(Use(v, SO) ∧ ¬Def(v, SO) ∧ ¬Use(v, GH)) ∨ (LocalCall(m, SO) ∧ ¬LocalCall(m, GH) ∧ ¬InstanceCall(m, GH))
Exception Handling	Insert/delete a try-catch block	(Insert(t_1, t_2, i) ∨ Delete(t_1)) ∧ NodeType(t_1, TryStatement)
	Insert/delete a thrown exception in a method header	Changed(t_1) ∧ NodeType(t_1, Type) ∧ Parent(t_2, t_1) ∧ NodeType(t_2, MethodDeclaration) ∧ NodeValue(t_1, t) ∧ isExceptionType(t)
	Update the exception type	Update(t_1, t_2) ∧ NodeType(t_1, SimpleType) ∧ NodeType(t_2, SimpleType) ∧ NodeValue(t_1, v_1) ∧ isExceptionType(v_1) ∧ NodeValue(t_2, v_2) ∧ isExceptionType(v_2)
	Change statements in a catch block	Changed(t_1) ∧ Ancestor(t_2, t_1) ∧ NodeType(t_2, CatchClause)
	Change statements in a finally block	Changed(t_1) ∧ Ancestor(t_2, t_1) ∧ NodeType(t_2, FinallyBlock)
Logic Customization	Change a method call	Changed(t_1) ∧ Ancestor(t_2, t_1) ∧ NodeType(t_2, MethodInvocation)
	Update a constant value	Update(t_1, t_2) ∧ NodeType(t_1, Literal) ∧ NodeType(t_2, Literal)
	Change a conditional expression	Changed(t_1) ∧ Ancestor(t_2, t_1) ∧ (NodeType(t_2, IfCondition) ∨ NodeType(t_2, LoopCondition) ∨ NodeType(t_2, SwitchCase))
	Change the type of a variable	Update(t_1, t_2) ∧ NodeType(t_1, Type) ∧ NodeType(t_2, Type)
Refactoring	Rename a variable/field/method	Update(t_1, t_2) ∧ NodeType(t_1, Name)
	Replace hardcoded constant values with variables	Delete(t_1) ∧ NodeType(t_1, Literal) ∧Insert(t_1, t_2, i) ∧ NodeType(t_1, Name) ∧ Match(t_1, t_2)
	Inline a field	Delete(t_1) ∧ NodeType(t_1, Name) ∧Insert(t_1, t_2, i) ∧ NodeType(t_1, Literal) ∧ Match(t_1, t_2)
Miscellaneous	Change access modifiers	Changed(t_1) ∧ NodeType(t_1, Modifier) ∧ NodeValue(t_1, v) ∧ v ∈ {private, public, protected, static}
	Change a log/print statement	Changed(t_1) ∧ NodeType(t_1, MethodInvocation) ∧ NodeValue(t_1, m) ∧ isLogMethod(m)
	Style reformatting (i.e., inserting/deleting curly braces)	Changed(t_1) ∧ NodeType(t_1, Block) ∧ Parent(t_2, t_1) ∧ ¬Changed(t_2) ∧ Child(t_3, t_1) ∧ ¬Changed(t_3)
	Change Java annotations	Changed(t_1) ∧ NodeType(t_1, Annotation)
	Change code comments	Changed(t_1) ∧ NodeType(t_1, Comment)

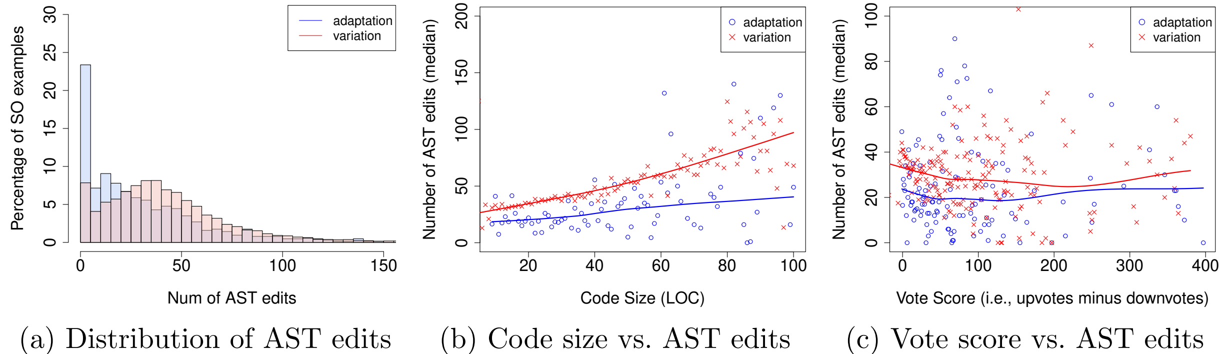
GumTree Edit Operation	Syntactic Predicate	Semantic Predicate
Insert(t_1, t_2, i) inserts a new tree node t_1 as the i-th child of t_2 in the AST of a GitHub snippet.	**NodeType**(t_1, X) checks if the node type of t_1 is X.	**Exception**(e, P) checks if e is an exception caught in a `catch` clause or thrown in a method header in program P.
	NodeValue(t_1, v) checks if the corresponding source code of node t_1 is v.	**LocalCall**(m, P) checks if m is a local method call in program P.
Delete(t) removes the tree node t from the AST of a SO example.	**Match**(t_1, t_2) checks if t_1 and t_2 are matched based on surrounding nodes regardless of node types.	**InstanceCall**(m, P) checks if m is an instance call in program P.
	Parent(t_1, t_2) checks if t_1 is the parent of t_2 in the AST.	**Def**(v, P) checks if variable v is defined in program P.
Update(t_1, t_2) updates the tree node t_1 in a SO example with t_2 in the GitHub counterpart.	**Ancestor**(t_1, t_2) checks if t_1 is the ancestor of t_2 in the AST.	**Use**(v, P) checks if variable v is used in program P.
	Child(t_1, t_2) checks if t_1 is the child of t_2.	**IsExceptionType**(X) checks if X contains "Exception".
Move(t_1, t_2, i) moves an existing node t_1 in the AST of a SO example as the i-th child of t_2 in the GitHub counterpart.	**Changed**(t_1) is a shorthand for **Insert**(t_1, t_2, i) ∨ **Delete**(t_1) ∨ **Update**(t_1, t_2) ∨ **Move**(t_1, t_2), which checks any edit operation on t_1.	**IsLogMethod**(X) checks if X is one of the predefined log methods, e.g., log, println, error, etc.
		IsCleanMethod(X) checks if X is one of the predefined resource clean-up methods, e.g., close, recycle, dispose, etc.

initially inspected 90 samples from each dataset and had already observed convergent adaptation types. We continued to inspect more and stopped after inspecting 200 samples from each dataset, since the list of adaptation types was converging. This is a typical procedure in qualitative analysis [24]. The two authors then discussed with the other authors to refine the adaptation types. Finally, we built a taxonomy of 24 adaptation types in 6 high-level categories, as shown in Table 5.1.

Code Hardening. This category includes four adaptation types that strengthen SO examples in a target project. *Insert a conditional* adds an `if` statement that checks for corner cases or protects code from invalid input data such as `null` or an out-of-bound index. *Insert a final modifier* enforces that a variable is only initialized once and the assigned value or reference is never changed, which is generally recommended for clear design and better performance due to static inlining. *Handle a new exception* improves the reliability of a code example by handling any missing exceptions, since exception handling is often omitted in examples in SO [152]. *Clean up unmanaged resources* helps release unneeded resources such as file streams and web sockets to avoid resource leaks [129].

Resolve Compilation Errors. SO examples are often incomplete with undefined variables and method calls [42, 149]. *Declare an undeclared variable* inserts a statement to declare an unknown variable. *Specify a target of method invocation* resolves an undefined method call by specifying the receiver object of that call. In an example about getting CPU usage [6], one comment complains the example does not compile due to an unknown method call, `getOperatingSystemMXBean`. Another suggests to preface the method call with an instance, `ManagementFactory`, which is also evidenced by its GitHub counterpart [16]. Sometimes, statements that use undefined variables and method calls are simply deleted.

Exception Handling. This category represents changes of the exception handling logic in `catch/finally` blocks and `throws` clauses. One common change is to customize the actions in a `catch` block, e.g., printing a short error message instead of the entire stack trace.



(a) Distribution of AST edits (b) Code size vs. AST edits (c) Vote score vs. AST edits

Figure 5.2: Code size (LOC) and vote scores on the number of AST edits in a SO example

Some developers handle exceptions locally rather than throwing them in method headers. For example, while the SO example [10] throws a generic `Exception` in the `addLibraryPath` method, its GitHub clone [14] enumerates all possible exceptions such as `SecurityException` and `IllegalArgumentException` in a `try-catch` block. By contrast, propagating the exceptions to upstream by adding `throws` in the method header is another way to handle the exceptions.

Logic Customization. Customizing the functionality of a code example to fit a target project is a common and broad category. We categorize logic changes to four basic types. *Change a method call* includes any edits in a method call, e.g., adding or removing a method call, changing its arguments or receiver, etc. *Update a constant value* changes a constant value such as the thread sleep time to another value. *Change a conditional expression* includes any edits on the condition expression of an `if` statement, a `loop`, or a `switch case`.

Update a type name replaces a variable type or a method return type with another type. For example, `String` and `StringBuffer` appear in multiple SO examples, and a faster type, `StringBuilder`, is used in their GitHub clones instead. Such type replacement often involves extra changes such as updating method calls to fit the replaced type or adding method calls to convert one type to another. For example, instead of returning `InetAddress` in a SO example [9], its GitHub clone [12] returns `String` and thus converts the IP address object to its string format using a new `Formatter` API.

Refactoring. 31% of inspected GitHub counterparts use a method or variable name different from the SO example. Instead of `slider` in a SO example [7], `timeSlider` is used in one GitHub counterpart [13] and `volumnSlider` is used in another counterpart [11]. Because SO examples often use hardcoded constant values for illustration purposes, GitHub counterparts may use variables instead of hardcoded constants. However, sometimes, a GitHub counterpart such as [15] does the opposite by inlining the values of two constant fields, `BUFFERSIZE` and `KB`, since these fields do not appear along with the copied method, `downloadWithHttpClient` [8].

Miscellaneous. Adaptation types in this category do not have a significant impact on the reliability and functionality of a SO example. However, several interesting cases are still worth noting. In 91 inspected examples, GitHub counterparts include comments to explain the reused code. Sometimes, annotations such as `@NotNull` or `@DroidSafe` appear in GitHub counterparts to document the constraints of code.

5.4.2 Automated Adaptation Categorization

Based on the manual inspection, we build a rule-based classification technique that automatically categorizes AST edit operations generated by GumTree to different adaptation types. GumTree supports four edit operations—**insert**, **delete**, **update**, and **move**, described in Column GumTree Edit Operation in Table 5.1. Given a set of AST edits, our technique leverages both syntactic and semantic rules to categorize the edits to 24 adaptation types. Column Rule in Table 5.1 describes the implementation logic of categorizing each adaptation type.

Syntactic-based Rules. 16 adaptation types are detected based on syntactic information, e.g., edit operation types, AST node types and values, etc. Column Syntactic Predicate defines such syntactic information, which is obtained using the built-in functions provided

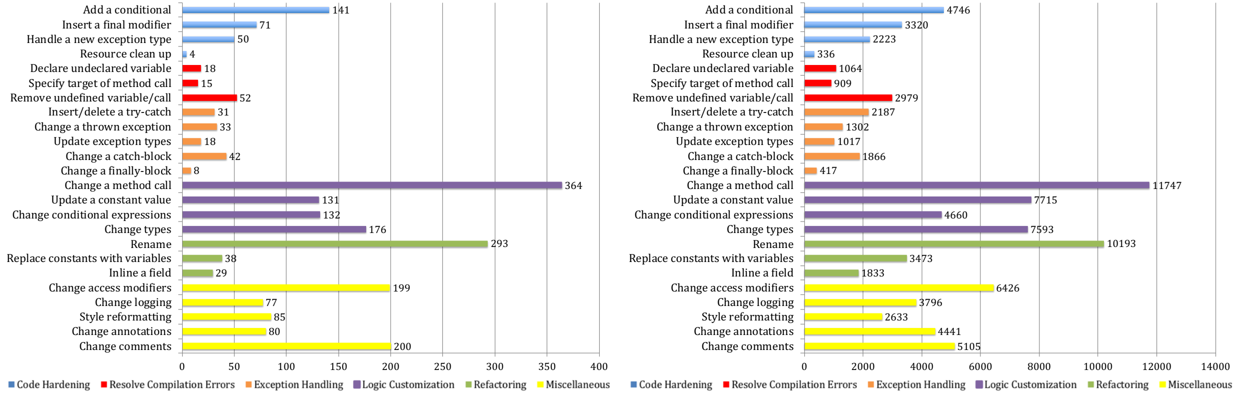
by GumTree. For example, the rule *insert a final modifier* checks for an edit operation that inserts a `Modifier` node whose value is `final` in a GitHub clone.

Semantic-based Rules. 8 adaptation types require leveraging semantic information to be detected (Column Semantic Predicate). For example, the rule *declare an undeclared variable* checks for an edit operation that inserts a `VariableDeclaration` node in the GitHub counterpart and the variable name is *used* but not *defined* in the SO example. Our technique traverses ASTs to gather such semantic information. For example, our AST visitor keeps track of all declared variables when visiting a `VariableDeclaration` AST node, and all used variables when visiting a `Name` node.

5.4.3 Accuracy of Adaptation Categorization

We randomly sampled another 100 SO examples and their GitHub clones to evaluate our automated categorization technique. To reduce bias, the second author who was not involved in the previous manual inspection labeled the adaptation types in this validation set. The ground truth contains 449 manually labeled adaptation types in 100 examples. Overall, our technique infers 440 adaptation types with 98% precision and 96% recall. In 80% of SO examples, our technique infers all adaptation types correctly. In another 20% of SO examples, it infers some but not all expected adaptation types.

Our technique infers incorrect or missing adaptation types for two main reasons. First, our technique only considers 24 common adaptation types in Table 5.1 but does not handle infrequent ones such as refactoring using lambda expressions and rewriting `++i` to `i++`. Second, GumTree may generate sub-optimal edit scripts with unnecessary edit operations in about 5% of file pairs, according to [45]. In such cases, our technique may mistakenly report incorrect adaptation types.



(a) Adaptations: 629 explicitly attributed SO examples

(b) Variations: 21,207 potentially reused SO examples

Figure 5.3: Frequencies of categorized adaptation types in two datasets

5.5 Empirical Study

5.5.1 How many edits are potentially required to adapt a SO example?

We apply the adaptation categorization technique to quantify the extent of adaptions and variations in the two datasets. We measure AST edits between a SO example and its GitHub counterpart. If a SO code example has multiple GitHub counterparts, we use the average number. Overall, 13,595 SO examples (96%) in the variation dataset include a median of 39 AST edits (mean 47). 556 SO examples (88%) in the adaptation dataset include a median of 23 AST edits (mean 33). Figure 5.2a compares the distribution of AST edits in these two datasets. In both datasets, most SO examples have variations from their counterparts, indicating that integrating them to production code may require some type of adaptations.

Figure 5.2b shows the median number of AST edits in SO examples with different lines of code. We perform a non-parametric local regression [119] on the example size and the number of AST edits. As shown by the two lines in Figure 5.2b, there is a strong positive correlation between the number of AST edits and the SO example size in both datasets—long

SO examples have more adaptations than short examples.

Stack Overflow users can vote a post to indicate the applicability and usefulness of the post. Therefore, votes are often considered as the main quality metric of SO examples [89]. Figure 5.2c shows the median number of AST edits in SO examples with different vote scores. Although the adaptation dataset has significantly higher votes than the variation dataset (Figure 5.2c), there is no strong positive or negative correlation between the AST edit and the vote score in both sets. This implies that highly voted SO examples do not necessarily require fewer adaptations than those with low vote scores.

5.5.2 What are common adaptation and variation types?

Figure 5.3 compares the frequencies of the 24 categorized adaptation types (Column Adaptation Type in Table 5.1) for the adaptation and variation datasets. If a SO code example has multiple GitHub counterparts, we only consider the distinct types among all GitHub counterparts to avoid the inflation caused by repetitive variations among different counterparts. The frequency distribution is consistent in most adaptation types between the two datasets, indicating that *variation patterns resemble adaptation patterns*. Participants in the user study also appreciate being able to see variations in similar GitHub code, since "it highlights the best practices followed by the community and prioritizes the changes that I should make first," as P5 explained.

In both datasets, the most frequent adaptation type is *change a method call* in the logic customization category. Other logic customization types also occur frequently. This is because SO examples are often designed for illustration purposes with contrived usage scenarios and input data, and thus require further logic customization. *Rename* is the second most common adaptation type. It is frequently performed to make variable and method names more readable for the specific context of a GitHub counterpart. 35% and 14% of SO examples

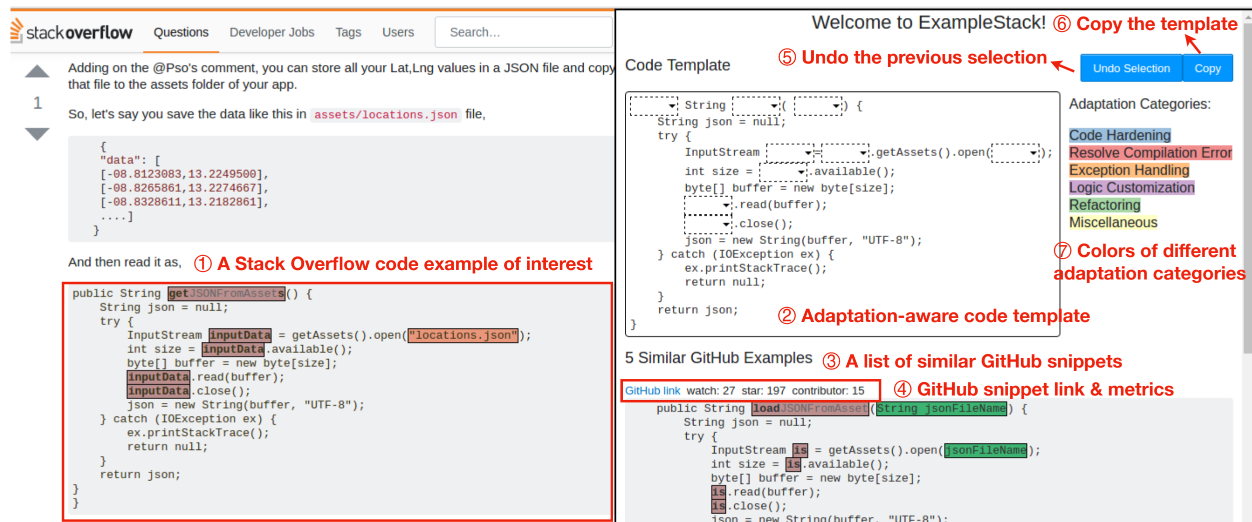


Figure 5.4: In the lifted template, common unchanged code is retained, while adapted regions are abstracted with *hot spots*.

in the variation dataset and the adaptation dataset respectively include undefined variables or local method calls, leading to compilation errors. The majority of these compilation errors (60% and 61% respectively) could be resolved by simply removing the statements using these undefined variables or method calls. 34% and 22% of SO examples in the two datasets respectively include new conditionals (e.g., an `if` check) to handle corner cases or reject invalid input data.

To understand whether the same type of adaptations appears repetitively on the same SO example, we count the number of adaptation types shared by different GitHub counterparts. Multiple clones of the same SO example share at least one same adaptation type in the 70% of the adaptation dataset and 74% of the variation dataset. In other words, *the same type of adaptations is recurring among different GitHub counterparts.*

5.6 Conclusion

This paper provides a comprehensive analysis of common adaptation and variation patterns of online code examples by both overapproximating and underapproximating reused code from Stack Overflow to GitHub. Our quantitative analysis shows that the same type of adaptations and variations appears repetitively among different GitHub clones of the same SO example, and variation patterns resemble adaptation patterns. This implies that different GitHub developers may apply similar adaptations to the same example over and over again independently.

Chapter 6

Statement-level Recommendation for Related Code

The material in this chapter is part of the following paper, and is included here with permission from ACM.

S. Luan, D. Yang, C. Barnaby, K. Sen and S. Chandra. Aroma: Code Recommendation via Structural Code Search. In Proceedings of the ACM on Programming Languages, Volume 3(OOPSLA), Oct 2019.

This work is done during my internship with Facebook in summer 2018. The paper proposed a statement-level code recommendation tool called Aroma. I contributed to the work by collecting all the data needed code search base and evaluation, implementing the clustering and intersection algorithm and part of the searching algorithm, and evaluating the recommendation results. In this chapter, I only included the parts that I was involved in.

6.1 Introduction

Suppose an Android programmer wants to write code to decode a bitmap. The programmer is familiar with the libraries necessary to write the code, but they are not quite sure how to write the code completely with proper error handling and suitable configurations. They write the code snippet shown in Listing 6.1 as a first attempt. The programmer now wants to know how others have implemented this functionality fully and correctly in related projects. Specifically, they want to know what is the customary way to extend the code so that proper setup is done, common errors are handled, and appropriate library methods are called. It would be nice if a tool could return a few code snippets shown in Listings 6.2, 6.3, which demonstrate how to configure the decoder to use less memory, and how to handle potential runtime exceptions, respectively. We call this the *code recommendation problem.*

```
1 InputStream input = manager.open(fileName);
2 Bitmap image = BitmapFactory.decodeStream(input);
```

Listing 6.1: Suppose an Android programmer writes this code to decode a bitmap.

```
1 final BitmapFactory.Options options = new BitmapFactory.Options();
2 options.inSampleSize = 2;
3 Bitmap bmp = BitmapFactory.decodeStream(is, null, options);
```

Listing 6.2: A recommended code snippet that shows how to configure the decoder to use less memory. Recommended lines are highlighted.[1]

```
1 try {
2 InputStream is = am.open(fileName);
3 image = BitmapFactory.decodeStream(is);
4 is.close();
5 } catch (IOException e) {
6 // ...
7 }
```

[1] Adapted from https://github.com/zom/Zom-Android/blob/master/app/src/main/java/org/awesomeapp/messenger/ui/stickers/StickerGridAdapter.java#L67. Accessed in August 2018.

Listing 6.3: Another recommended code snippet that shows how to properly close the input stream and handle any potential `IOException`. Recommended lines are highlighted.[2]

There are a few existing techniques which could potentially be used to get code recommendations. For example, code-to-code search tools [68, 73] could retrieve relevant code snippets from a corpus using a partial code snippet as query. However, such code-to-code search tools return lots of relevant code snippets without removing or aggregating similar-looking ones. Moreover, such tools do not make any effort to carve out common and concise code snippets from similar-looking retrieved code snippets. Pattern-based code completion tools [87, 91, 93] mine common API usage patterns from a large corpus and use those patterns to recommend code completion for partially written programs as long as the partial program matches a prefix of a mined pattern. Such tools work well for the mined patterns; however, they cannot recommend any code outside the mined patterns—the number of mined patterns are usually limited to a few hundreds. We emphasize that the meaning of the phrase "code recommendation" in Aroma is different from the term "API code recommendation" [90, 94]. The latter is a recommendation engine for the next API method to invoke given a code change, whereas Aroma aims to recommend code snippets, as shown in Listings 6.2, 6.3, for programmers to learn common usages and integrate those usages with their own code. Aroma's recommendations contain more syntactic variety than just API usages; for instance, the recommended code snippet in Listing 6.3 includes a `try-catch` block, and Example B in Table 6.1 recommends adding an `if` statement that modifies a variable. Code clone detectors [38, 64, 66, 116] are another set of techniques that could potentially be used to retrieve recommended code snippets. However, code clone detection tools usually retrieve code snippets that are almost identical to a query snippet. Such retrieved code snippets may not always contain extra code

[2] Adapted from `https://github.com/yuyuyu123/ZCommon/blob/master/zcommon/src/main/java/com/cc/android/zcommon/utils/android/AssetUtils.java#L37`. Accessed in August 2018.

which could be used to extend the query snippet.

We propose AROMA, a code recommendation engine. Given a code snippet as input query and a large corpus of code containing millions of methods, AROMA returns a set of recommended code snippets such that each recommended code snippet:

- contains the query snippet approximately, and
- is contained approximately in a non-empty set of method bodies in the corpus.

Furthermore, AROMA ensures that any two recommended code snippets are not quite similar to each other.

AROMA works by first indexing the given corpus of code. Then AROMA searches for a small set (e.g. 1000) of method bodies which contain the query code snippet *approximately*.

A challenge in designing this search step is that a query snippet, unlike a natural language query, has structure, which should be taken into account while searching for code. Once AROMA has retrieved a small set of code snippets which approximately contain the query snippet, AROMA prunes the retrieved snippets so that the resulting pruned snippets become similar to the query snippet. It then ranks the retrieved code snippets based on the similarity of the pruned snippets to the query snippet. This step helps to rank the retrieved snippets based on how well they contain the query snippet. The step is precise, but is relatively expensive; however, the step is only performed on a small set of code snippets, making it efficient in practice. After ranking the retrieved code snippets, AROMA clusters the snippets so that similar snippets fall under the same cluster. AROMA then intersects the snippets in each cluster to carve out a maximal code snippet which is common to all the snippets in the cluster and which contains the query snippet. The set of intersected code snippets are then returned as recommended code snippets. Figure 6.3 shows an outline of the algorithm. For the query shown in Listing 6.1, AROMA recommends the code snippets shown in Listings 6.2,

6.3. The right column of Table 6.1 shows more examples of code snippets recommended by Aroma for the code queries shown on the left column of the table.

To our best knowledge, Aroma is the first tool which could recommend relevant code snippets given a query code snippet. The advantages of Aroma are the following:

- A code snippet recommended by Aroma does not simply come from a single method body, but is generated from several similar-looking code snippets via intersection. This increases the likelihood that Aroma's recommendation is idiomatic rather than one-off.
- Aroma does not require mining common coding patterns or idioms ahead of time. Therefore, Aroma is not limited to a set of mined patterns—it can retrieve new and interesting code snippets on-the-fly.
- Aroma is fast enough to use in real time. A key innovation in Aroma is that it first retrieves a small set of snippets based on approximate search, and then performs the heavy-duty pruning and clustering operations on this set. This enables Aroma to create recommended code snippets on a given query from a large corpus containing millions of methods within a couple of seconds on a multi-core server machine.
- Aroma is easy to deploy for different programming languages because its core algorithm works on generic parse trees. We have implemented Aroma for Hack, Java, JavaScript and Python.
- Although we developed Aroma for the purpose of code recommendation, it could be used to also perform efficient and precise code-to-code structural search.

We have implemented Aroma in C++ for four programming languages: Hack [141], Java, JavaScript and Python. We have also implemented IDE plugins for all of these four languages. We report our experimental evaluation of Aroma for the Java programming language. We have used Aroma to index 5,417 GitHub Java Android projects. We performed

Table 6.1: Aroma code recommendation examples

Query Code Snippet	Aroma Code Recommendation with Extra Lines Highlighted

Query Code Snippet

```
1 TextView textView = (TextView) view.findViewById(R.id.textview);
2 SpannableString content = new SpannableString("Content");
3 content.setSpan(new UnderlineSpan(), 0, content.length(), 0);
4 textView.setText(content);
```

Example A: Configuring Objects.

- This code snippet adds underline to a piece of text.[1]
- The recommended code suggests adding a callback handler to pop up a dialog once the underlined text is touched upon.
- **Intersected from a cluster of 2 methods.**[2]

Aroma Code Recommendation

```
1 TextView licenseView = (TextView)
        findViewById(R.id.library_license_link);
2 SpannableString underlinedLicenseLink = new SpannableString(
3     getString(R.string.library_license_link));
4 underlinedLicenseLink.setSpan(new UnderlineSpan(), 0,
        underlinedLicenseLink.length(), 0);
5 licenseView.setText(underlinedLicenseLink);
6 licenseView.setOnClickListener(v -> {
7     FragmentManager fm = getSupportFragmentManager();
8     LibraryLicenseDialog libraryLicenseDlg = new
            LibraryLicenseDialog();
9     libraryLicenseDlg.show(fm, "fragment_license"); });
```

Query Code Snippet

```
1 Bitmap bitmap = BitmapFactory.decodeResource(getResources(),
        R.drawable.image);
```

Example B: Post-Processing.

- This code snippet decodes a bitmap.[3]
- The recommended code suggests applying Gaussian blur on the decoded image, a customary effect to be applied.
- **Intersected from a cluster of 4 methods.**[4]

Aroma Code Recommendation

```
1 int radius = seekBar.getProgress();
2 if (radius < 1) {
3     radius = 1;
4 }
5 Bitmap bitmap = BitmapFactory.decodeResource(getResources(),
        R.drawable.image);
6 imageView.setImageBitmap(blur.gaussianBlur(radius, bitmap));
```

Query Code Snippet

```
1 EditText et = (EditText)findViewById(R.id.inbox);
2 et.setSelection(et.getText().length());
```

Example C: Correlated Statements.

- This code snippet moves the cursor to the end in a text area.[5]
- The recommended code suggests also configuring the action bar to create a more focused view.
- **Intersected from a cluster of 2 methods.**[6]

Aroma Code Recommendation

```
1 super.onCreate(savedInstanceState);
2 setContentView(R.layout.material_edittext_activity_main);
3 getSupportActionBar().setDisplayHomeAsUpEnabled(true);
4 getSupportActionBar().setDisplayShowTitleEnabled(false);
5 EditText singleLineEllipsisEt = (EditText)
        findViewById(R.id.singleLineEllipsisEt);
6 singleLineEllipsisEt.setSelection(
        singleLineEllipsisEt.getText().length());
```

Query Code Snippet

```
1 PackageInfo pInfo =
        getPackageManager().getPackageInfo(getPackageName(), 0);
2 String version = pInfo.versionName;
```

Example D: Exact Recommendations.

- This partial code snippet gets the current version of the application. The rest of the code snippet (not shown) catches and handles possible `NameNotFound` errors.[7]
- The recommended code suggests the exact same error handling as in the original code snippet.
- **Intersected from a cluster of 2 methods.**[8]

Aroma Code Recommendation

```
1 try {
2     PackageInfo pInfo =
            getPackageManager().getPackageInfo(getPackageName(), 0);
3     String version = pInfo.versionName;
4     TextView versionView = (TextView)
            findViewById(R.id.about_project_version);
5     versionView.setText("v" + version);
6 } catch (PackageManager.NameNotFoundException ex) {
7     Log.e(...);
8 }
```

Query Code Snippet

```
1 i.putExtra("parcelable_extra", (Parcelable) myParcelableObject);
```

Example E: Alternative Recommendations.

- This partial code snippet demonstrates one way to attach an object to an `Intent`. The rest of the code snippet (not shown) shows a different way to serialize and attach an object.[9]
- **Intersected from a cluster of 10 methods.**[10]

Aroma Code Recommendation

```
1 Intent intent = new Intent(this, BoardTopicActivity.class);
2 intent.putExtra(SMTHApplication.BOARD_OBJECT, (Parcelable) board);
3 startActivity(intent);
```

- The recommended code does not suggest the other way of serializing the object, but rather suggests a common way to complete the operation by starting an activity with an `Intent` containing a serialized object.

[1] Adapted from the Stack Overflow post "Can I underline text in an android layout?" [https://stackoverflow.com/questions/2394939], by Anthony Forloney [https://stackoverflow.com/users/166712].
[2] Adapted from https://github.com/tonyvu2014/android-shoppingcart/blob/master/demo/src/main/java/com/android/tonyvu/sc/demo/ProductActivity.java.
[3] Adapted from the Stack Overflow post "How to set a bitmap from resource" [https://stackoverflow.com/questions/4955305], by xandy [https://stackoverflow.com/users/109112].
[4] Adapted from https://github.com/TonnyL/GaussianBlur/blob/master/app/src/main/java/io/github/marktony/gaussianblur/MainActivity.java.
[5] Adapted from the Stack Overflow post "Place cursor at the end of text in EditText" [https://stackoverflow.com/questions/6624186], by Marqs [https://stackoverflow.com/users/400493].
[6] Adapted from https://github.com/cymcsg/UltimateAndroid/blob/master/deprecated/UltimateAndroidGradle/demoofui/src/main/java/com/marshalchen/common/demoofui/sampleModules/MaterialEditTextActivity.java.
[7] Adapted from the Stack Overflow post "How to get the build/version number of your Android application?" [https://stackoverflow.com/questions/6593822], by plus- [https://stackoverflow.com/users/709635].
[8] Adapted from https://github.com/front-line-tech/background-service-lib/blob/master/SampleService/servicelib/src/main/java/com/flt/servicelib/AbstractPermissionExtensionAppCompatActivity.java.
[9] Adapted from the Stack Overflow post "How to send an object from one Android Activity to another using Intents?" [https://stackoverflow.com/questions/2141166], by Jeremy Logan [https://stackoverflow.com/users/76835].
[10] Adapted from https://github.com/zfdang/zSMTH-Android/blob/master/app/src/main/java/com/zfdang/zsmth_android/MainActivity.java.
All Stack Overflow content is licensed under CC-BY-SA 3.0. All URLs are accessed in August 2018.

our experiments for Android Java because we initially developed Aroma for Android based on internal developers' need. We evaluated Aroma using code snippets obtained from Stack Overflow. We manually analyzed and categorized the recommendations into several representative categories. We also evaluated Aroma recommendations on 50 partial code snippets, where we found that Aroma can recommend the exact code snippets for 37 queries, and in the remaining 13 cases Aroma recommends alternative recommendations that are still useful. On average, Aroma takes 1.6 seconds to create recommendations for a query code snippet on a 24-core CPU. In our large-scale automated evaluation, we used a micro-benchmarking suite containing artificially created query snippets to evaluate the effectiveness of various design choices in Aroma.

The rest of the paper is organized as follows: Section 6.2 presents a case study that reveals the opportunity for a code recommendation tool like Aroma. In Section 6.4, we describe the algorithm Aroma uses to create code recommendations. In Section 6.5 we manually assess how useful Aroma code recommendations are. Since code search is a key component of creating recommendations, in Section 6.6 we measure the search recall of Aroma and compare it with other techniques. Section 7.2 presents related work. Finally, Section 6.7 concludes the paper.

6.2 The Opportunity for Aroma

Aroma is based on the idea that new code often resembles code that has already been written— therefore, programmers can benefit from recommendations from existing code. To substantiate this claim, we conducted an experiment to measure the similarity of new code to existing code. This experiment was conducted on a large codebase in the Hack language.

We first collected all code commits submitted in a two-day period. From these commits,

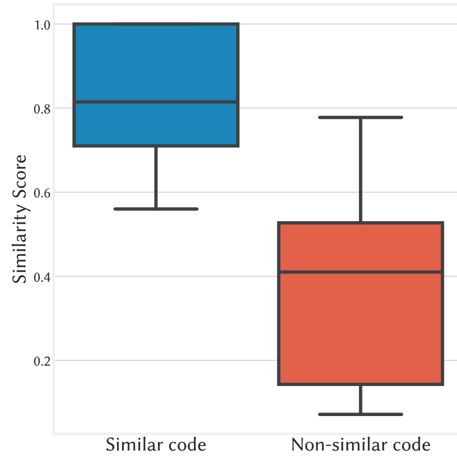


Figure 6.1: Distribution of similarity scores used to obtain threshold

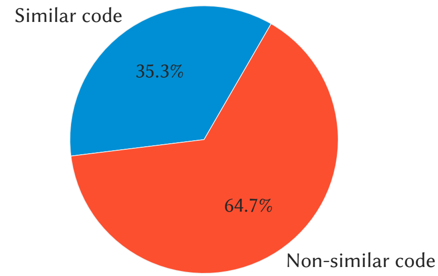


Figure 6.2: Proportion of new code similar to existing code

we extracted a set of changesets. A changeset is defined as a set of contiguous added or modified lines in a code commit. We filtered out changesets that were shorter than two lines or longer than seven lines. We decided to use this filter because longer changesets are more likely to span multiple methods, and we wanted to limit our dataset to code added or modified within a single method. Alternatively, we could have taken portions of changesets found within a single method; however, since changesets are raw text, finding the method boundaries involves additional parsing. We stuck to the simple solution of taking short changesets.

For each of the first 1000 changesets in this set, we used Aroma to perform a code-to-code search, taking the snippet as input and returning a list of methods in the repository that contain structurally similar code. Aroma was used because it was already implemented for Hack—but for the purpose of this experiment, any code-to-code search tool or clone detector can work. The results are ranked by similarity score: the percentage of features in the search query that are also found in the search result. For each changeset, we took the top-ranked method and its similarity score. 71 changesets did not yield any search result, because they contained only comments or variable lists, which Aroma disregards in search (see Section 6.4.2).

To interpret the results, we first needed to assess the correlation between the similarity score (i.e. a measure of the syntactic similarity) and the semantic similarity between the changeset and the result. Two authors manually looked over a random sample of 50 pairs of changesets and result methods, and decided whether this method contained code similar enough to the changeset that a programmer could adopt the existing code (by copy-pasting or refactoring) with minimal changes. Using this criteria, each pair was deemed "similar" or "not similar". Conflicting judgments were cross-checked and re-assessed. As shown in the box plot in Figure 6.1, there is a clear distinction in similarity scores between the manually-labeled "similar" and "not similar" pairs. Note that in this figure, the top and bottom of the box represents the third and first quartile of the data. The lines extending above and below the box represent the maximum and minimum, and the line running through the box represents the median value.

We chose the first quartile of the similarity scores in the manually-labeled similar pairs—0.71—as the threshold similarity score to decide whether a retrieved code snippet contains meaningful semantic similarities to new code in the commit. We found that for 35.3% of changesets, the most similar result had a score of at least 0.71, meaning that in these cases it would be easy for a programmer to adapt the existing code with minimal efforts, should the code be provided to them.

These results indicate that a considerable amount of new code contains similarities to code that already exists in a large code repository. With Aroma, we aim to utilize this similar code to offer concise, helpful code recommendations to programmers. The amount of similar code in new commits suggests that Aroma has the potential to save a lot of programmers' time and effort.

6.3 Related Work

Code Search Engines

Code-to-code search tools like FaCoY [68] and Krugle [73] take a code snippet as query and retrieve relevant code snippets from the corpus. FaCoY aims to find semantically similar results for input queries. Given a code query, it first searches in a Stack Overflow dataset to find natural language descriptions of the code, and then finds related posts and similar code. While these code-to-code search tools retrieve similar code at different syntactic and semantic levels, they do not attempt to create concise recommendations from their search results. Further, most of these search engines cannot be instantiated on our code corpus, so we could not experimentally compare Aroma with these search engines. For instance, the code search engine FaCoY only provides a VM-based demo that is instantiated on a fixed corpus which is not available publicly. We were also unable to instantiate FaCoY on our corpus for a direct comparison. Most other open-source code search tools, including Krugle and searchcode.com, suffer from the same problem. Instead, we compared Aroma with two conventional code search techniques based on featurization and TF-IDF in Section 6.6.1, and found that Aroma's pruning-based search technique in Phase II outperforms both techniques.

Many efforts have been made to improve keyword-based code search [22, 34, 78, 79, 113]. CodeGenie [74] uses test cases to search and reuse source code; SNIFF [35] works by inlining API documentation in its code corpus. SNIFF also intersects the search results to provide recommendations, but only targets at resolving natural language queries. The clustering algorithm in SNIFF is limited and does not take structure into account. Two statements are considered similar if they are syntactically similar after replacing variable names with types. The intersection of two code snippets is the set of statements that appear in both snippets. Due to the strict definition of similarity, SNIFF cannot find large clusters that contain

approximately similar code snippets. Also, SNIFF uses the longest common subsequence algorithm, whose limitations we discuss in Section 6.4.3.2. MAPO [155] recommends code examples by mining and indexing associated API usage patterns. Portfolio [80] retrieves functions and visualizes their usage chains. CodeHow [77] augments the query with API calls which are retrieved from documentation to improve search results. CoCaBu [120] augments the query with structural code entities. A developer survey [114] reports the top reason for code search is to find code examples or related APIs, and tools have been created for this need. While these code search techniques focus on creating code examples based on keyword queries, they do not support code-to-code search and recommendation.

Clone Detectors

Clone detectors are designed to detect syntactically identical or highly similar code. SourcererCC [116] is a token-based clone detector targeting Type 1, 2, and 3 clones. Compared with other clone detectors that also support Type 3 clones, including NiCad [38], Deckard [64], and CCFinder [66], SourcererCC has high precision and recall and also scales to large-size projects. One may repurpose a clone detector to find similar code, but since it is designed for finding highly similar code rather than code that contains the query code snippet—as demonstrated in Section 6.6.1—its results are not suitable for code recommendation.

Recent clone detection techniques explored other research directions, from finding semantically similar clones [67, 68, 115, 145], to finding gapped clones [134] and gapped clones with a large number of edits (large-gapped clones) [142]. These techniques may excel in finding a particular type of clone, but they sacrifice the precision and recall for Type 1 to 3 clones.

Pattern Mining and Code Completion

Code completion can be achieved by different approaches—from extracting the structural context of the code to mining recent histories of editing [30, 59, 62, 109]. GraPacc [91] achieves pattern-oriented code completion by first mining graph-represented coding patterns using GrouMiner [93], then searching for input code to produce code completion suggestions. More recent work [90, 92, 94] improves code completion by predicting the next API call given a code change. Pattern-oriented code completion requires mining usage patterns ahead of time, and cannot recommend any code outside of the mined patterns, while Aroma does not require pattern mining and recommends snippets on the fly.

API Documentation Tools

More techniques exist for improving API documentations and examples. The work by [32] synthesizes API usage examples through data flow analysis, clustering and pattern abstraction. The work by [124] augments API documentations with up-to-date source code examples. MUSE [85] generates code examples for a specific method using static slicing. SWIM [105] synthesizes structured call sequences based on a natural language query. The work by [130] augments API documentation with insights from Stack Overflow. These tools are limited to API usages and do not generalize to structured code queries.

6.4 Algorithm

Figure 6.3 illustrates the overall architecture of Aroma. In order to generate code recommendations, Aroma must first featurize the code corpus. To do so, Aroma parses the body of each method in the corpus and creates its parse tree. It extracts a set of structural features from each parse tree. Then, given a query code snippet, Aroma runs the following

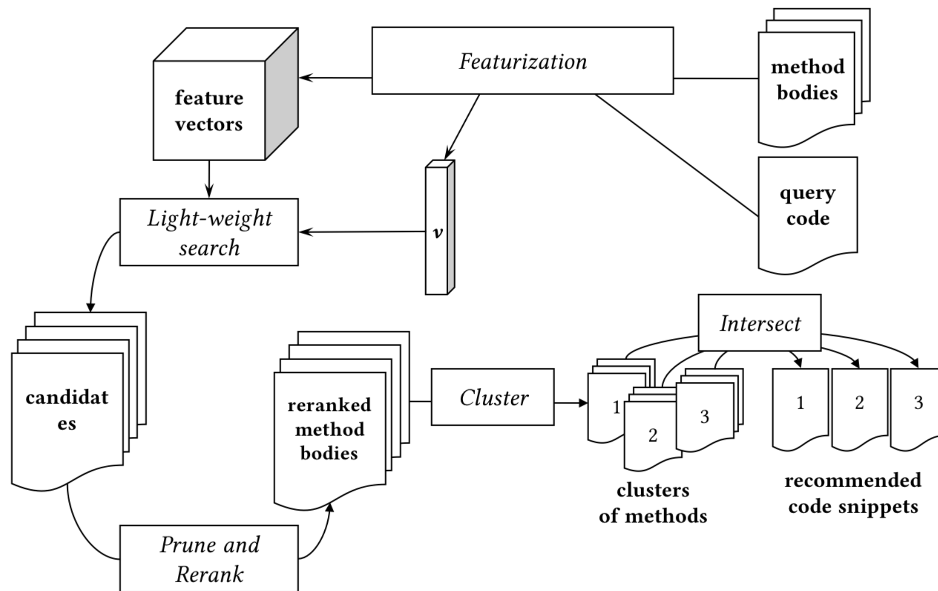


Figure 6.3: Aroma code recommendation pipeline

phases to create recommendations:

- **Light-weight Search.** In this phase, Aroma takes a query code snippet, and outputs a list of the top few (e.g. 1000) methods that have the most overlap with the query. To do so, Aroma extracts custom features from the query and each method in the corpus. Aroma intersects the set of features of the query and each method body, and uses the cardinality of the intersection to compute the degree of overlap between the query and the method body. To make this computation efficient, Aroma represents the set of features of a code snippet as a sparse vector and performs matrix multiplication to compute the degree of overlap of all methods with the query code.
- **Prune and Rerank.** In this phase, Aroma reranks the list of method bodies retrieved from the previous phase using a more precise, but expensive algorithm for computing similarity.
- **Cluster and Intersect.** In the final phase, Aroma clusters the reranked list of code snippets from the previous phase. Clustering is based on the similarity of the method

bodies. Clustering also needs to satisfy constraints which ensure that recommendations are of high quality. Therefore, we have devised a custom clustering algorithm which takes the constraints into account. AROMA then intersects the snippets in each cluster to come up with recommended code snippets. This approach of clustering and intersection helps to create a succinct, yet diverse set of recommendations.

We next describe the details of each step using the code snippet shown in Listing 6.4 as the running example.

```
1 if (view instanceof ViewGroup) {
2     for (int i = 0; i < ((ViewGroup) view).getChildCount(); i++) {
3         View innerView = ((ViewGroup) view).getChildAt(i);
4     }
5 }
```

Listing 6.4: A code snippet adapted from a Stack Overflow post.[3] This snippet is used as the running example through Section 6.4.

6.4.1 Definitions

In this section, we introduce several notations and definitions used to compute the features of a code snippet. The terminologies and notations are also used to describe AROMA formally.

Definition 1 (Keyword tokens) *This is the set of all tokens in a language whose values are fixed as part of the language. Keyword tokens include keywords such as `while`, `if`, `else`, and symbols such as {, }, `.`, `+`, `*`. The set of all keyword tokens is finite for a language.*

Definition 2 (Non-keyword tokens) *This is the set of all tokens that are not keyword tokens. Non-keyword tokens include variable names, method names, field names, and literals.*

[3]Adapted from the Stack Overflow post "How to hide soft keyboard on android after clicking outside EditText?" [https://stackoverflow.com/questions/11656129], by Navneeth G [https://stackoverflow.com/users/1135909]. CC-BY-SA 3.0 License. Accessed in August 2018.

Examples of non-keyword tokens are `i`, `length`, 0, 1, etc. The set of non-keyword tokens is non-finite for most languages.

Definition 3 (Simplified Parse Tree) *A simplified parse tree is a data structure we use to represent a program. It is recursively defined as a non-empty list whose elements could be any of the following:*

- *a non-keyword token,*
- *a keyword token, or*
- *a simplified parse tree.*

Moreover, a simplified parse tree cannot be a list containing a single simplified parse tree.

We picked this particular representation of programs instead of a conventional abstract syntax tree representation because the representation only consists of program tokens, and does not use any special language-specific rule names such as `IfStatement`, `block` etc. As such, the representation can be used uniformly across various programming languages. Moreover, one could perform an in-order traversal of a simplified parse tree and print the token names to obtain the original program, albeit unformatted. We use this feature of a simplified parse tree to show the recommended code snippets.

Definition 4 (Label of a Simplified Parse Tree) *The label of a simplified parse tree is obtained by concatenating all the elements of the list representing the tree as follows:*

- *If an element is a keyword token, the value of the token is used for concatenation.*
- *If an element is a non-keyword token or a simplified parse tree, the special symbol `#` is used for concatenation.*

For example, the label of the simplified parse tree `["x", ">", ["y", ".", "f"]]` is `"#>#"`.

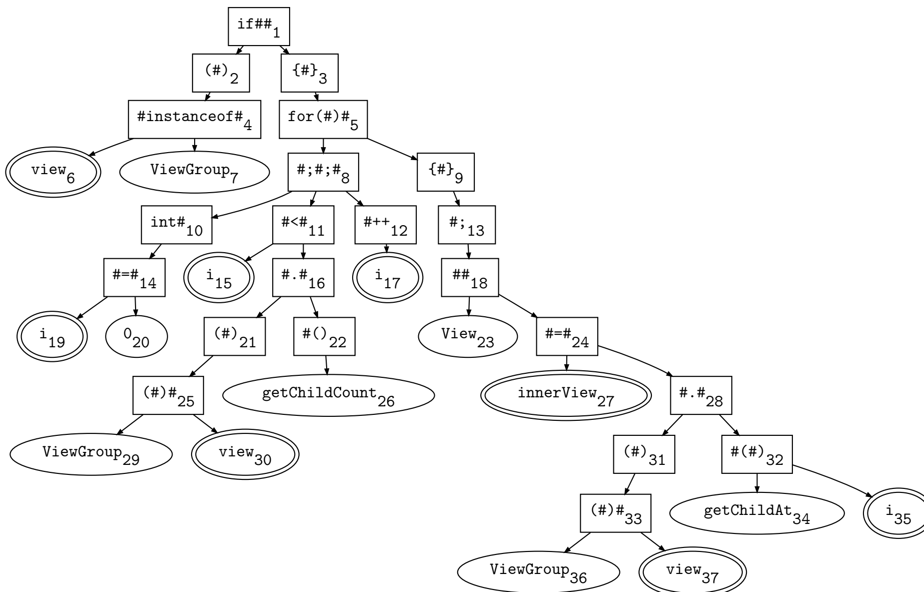


Figure 6.4: The simplified parse tree representation of the code in Listing 6.4. Keyword tokens at the leaves are omitted to avoid clutter. Variable nodes are highlighted in double circles.

Figure 6.4 visualizes the simplified parse tree of the code snippet in Listing 6.4. In the figure, each internal node represents a simplified parse tree, and is labeled using the tree's label as defined above. Since keyword tokens in a simplified parse tree become part of the label of the tree, we do not create leaf nodes for keyword tokens in the tree diagram—we only add leaf nodes for non-keyword tokens. We show the label of each node in the tree, and add a unique index to each label as subscript to distinguish between any two similar labels.

To obtain the simplified parse tree of a code snippet, Aroma relies on a language-specific parser. For example, Aroma utilizes the `ANTLR4` [99] Java parser to produce the parse tree for a Java program. Aroma traverses the parse tree produced by the parser to collect at each internal node the tokens and subtrees that are immediate descendants of each internal node of the parse tree. The collected elements at each node form a list, which is a simplified parse tree. Aroma uses the list at each internal node to create the label for the node. The distinction between keyword and non-keyword tokens is done using the language's lexical

specification. Aroma performs a second traversal of the tree and uses the static scoping rules of the language to identify the global variables. Aroma uses the knowledge of the global variables in the featurization phase which we describe later. Note that this process of creating a simplified parse tree from a code snippet is language-dependent and requires knowledge about the grammar and static scoping rules of the language. Once the simplified parse tree of a code snippet has been created, the rest of Aroma's algorithm is language-agnostic.

Given a simplified parse tree t, we use the following notations. All examples refer to Figure 6.4.

- $L(t)$ denotes the label of the tree t. E.g. $L(\texttt{if\#\#}_1) = \texttt{if\#\#}$.
- $N(t)$ denotes the list of all non-keyword tokens present in t or in any of its sub-trees, in the same order as appearing in the source code.

  E.g. $N(\texttt{\#.\#}_{28}) = [\texttt{ViewGroup}_{36}, \texttt{view}_{37}, \texttt{getChildAt}_{34}, \texttt{i}_{35}]$.
- If n is a non-keyword direct child of t, then we use $P(n)$ to denote the parent of n which is t. E.g. $P(\texttt{view}_6) = \texttt{\#instanceof\#}_4$.
- If t' is a simplified parse tree and is a direct child of t, then we again use $P(t')$ to denote the parent of t' which is t. E.g. $P(\texttt{\#instanceof\#}_4) = \texttt{(\#)}_2$.
- If n_1 and n_2 are two non-keyword tokens in a program and if n_2 appears after n_1 in the program without any intervening non-keyword token, then we use $\texttt{Prev}(n_2)$ to denote n_1 and $\texttt{Next}(n_1)$ to denote n_2. E.g. $\texttt{Prev}(\texttt{view}_{30}) = \texttt{ViewGroup}_{29}, \texttt{Next}(\texttt{ViewGroup}_{29}) = \texttt{view}_{30}$.
- If n_1 and n_2 are two non-keyword tokens denoting the same local variable in a program and if n_1 and n_2 are the two consecutive usages of the variable in the source code, then

we use $\texttt{PrevUse}(n_2)$ to denote n_1 and $\texttt{NextUse}(n_1)$ to denote n_2. E.g. $\texttt{PrevUse}(\texttt{view}_{30}) = \texttt{view}_6, \texttt{NextUse}(\texttt{view}_{30}) = \texttt{view}_{37}$.

- If n is a non-keyword token denoting a local variable and it is the ith child of its parent t, then the context of n, denoted by $C(n)$, is defined to be:
  - $(i, L(t))$, if $L(t) \neq \texttt{\#.\#}$. E.g. $C(\texttt{view}_{30}) = (2, \texttt{(\#)\#})$.
  - The first non-keyword token that is not a local variable in $N(t)$, otherwise. This is to accommodate for cases like `x.foo()`, where we want the context feature for `x` to be `foo` rather than $(1, \texttt{\#.\#})$, because the former better reflects its usage context.

6.4.2 Featurization

The high-level goal of this step is to take a simplified parse tree for a code snippet, and extract a set of structural features from that parse tree. A key requirement of the features is that if two code snippets are similar, they should have the same collection of features.

A simple way to featurize a code snippet is to treat the labels of all nodes in the simplified parse tree as features. This simple approach creates problem if we have two code snippets 1) which only differ in local variable names, and 2) where one code snippet can be obtained from the other by alpha renaming the local variables. The two code snippets should be considered as similar, but the collection of features will differ in the name of some of the variables. Therefore, we replace each token that denotes a local variable by a special token `#VAR`. We do not perform similar replacements for global variables and method names. This is because such identifiers are often part of some library API and cannot be alpha-renamed to obtain similar programs.

Treating the labels of parse tree nodes as the only features does not help to capture the relation between the nodes. Such relations are necessary to identify the structural features

of a code snippet. For example, without such a relation Aroma will treat the snippets `if (x > 0) z = 3;` and `if (z > 3) x = 0;` as similar since they have the exact same collection of node labels (i.e. {`if##, (#), #>#, #;, #=#, 0, 3, #VAR, #VAR`}). If we can somehow create a feature encapsulating the fact that `3` belongs to the body of the first `if` statement, Aroma will distinguish between the two snippets. Therefore, Aroma also creates features which represent some relations between certain pairs of nodes in the parse tree. Examples of some such features involving the token 3 are `(if##,2,3)`, `(#=#,2,3)`, and `(#VAR,3)`. The first feature, which is denoted as a triplet, states that the 2nd child of a node labeled `if##` has a descendant leaf node with label `3`. Similarly, the second feature asserts that the 2nd child of a node labeled `#=#` has a descendant leaf node with label `3`. We call these two features *parent features*, as they help capture the relation of a leaf node with its parent, grand-parent, and great-grand parent. The third feature relays the fact that a variable leaf node appears before 3. We call such features *sibling features.* In summary, the parent features and sibling features capture some local relations between the nodes in a parse tree. However, these features are not exhaustive enough to recreate the parse tree from the features. These non-exhaustiveness of features helps Aroma tolerate some non-similarities in otherwise similar code snippets, and helps Aroma to retrieve some closely related, but nonidentical code snippets during search.

Since we replace all local variable names with `#VAR`, we also need to relate two variable usages in a code snippet which refer to the same local variable. For example, in the code snippet `if (y < 0) x = -x;`, we will have three features of the form `#VAR` corresponding to the two occurrences of `x` and one occurrence of `y`. However, the collection of features described so far does not express the fact that two `#VAR` features refer to the same variable. There is no direct way to state two variables are related since we have gotten rid of variable names. Rather, we capture features about the fact that the *consecutive usage context* of the same local variables are related. We call such features *variable usage features.* For example, the context of the two usages of `x` are `(1,#=#)` and `(1,-#)`, respectively. The first context

corresponds to the first usage of `x` and denotes that there is a variable which is the first child of the node labeled `#=#`. The index and the parent node label together forms the context of this particular variable usage. Similarly, the second context denotes the second usage of `x`. We create a feature of the form `((1,#=#),(1,-#))` which captures the relation between the context of two consecutive usage of the same variable.

We now describe formally how a code snippet is featurized by AROMA. Given a simplified parse tree, we extract four kinds of features for each non-keyword token n in the program represented by the tree:

1. A *Token Feature* of the form n. If n is a local variable, we replace n with `#VAR`.
2. *Parent Features* of the form $(n, i_1, L(t_1))$, $(n, i_2, L(t_2))$, and $(n, i_3, L(t_3))$. Here n is the i_1^{th} child of t_1, t_1 is the i_2^{th} child of t_2, and t_2 is the i_3^{th} child of t_3. As before, if n is a local variable, then we replace n with `#VAR`. Note that in each of these features, we do not specify if the third element in a feature is the parent, grand-parent, or the great-grand parent. This helps AROMA to tolerate some non-similarities in otherwise similar code snippets.
3. *Sibling Features* of the form $(n, \texttt{Next}(n))$ and $(\texttt{Prev}(n), n)$. As before, if any of $n, \texttt{Next}(n), \texttt{Prev}(n)$ is a local variable, it is replaced with `#VAR`.
4. *Variable Usage Features* of the form $(C(\texttt{PrevUse}(n)), C(n))$ and $(C(n), C(\texttt{NextUse}(n)))$. We only add these features if n is a local variable.

For a non-keyword token $n \in N(t)$, we use $F(n)$ to denote the multi-set of features extracted for n. We extend the definition of F to a set of non-keyword tokens Q as follows: $F(Q) = \biguplus_{n \in Q} F(n)$ where $\uplus$ denotes multi-set union. For a simplified parse tree t, we use $F(t)$ to denote the multi-set of features of all non-keyword tokens in t, i.e. $F(t) = F(N(t))$. Let $\mathcal{F}$ be the set of all features that can extracted from a given corpus of code.

Table 6.2 illustrates the features extracted for two non-keyword tokens from the simplified parse tree in Figure 6.4. In the interest of space, we do not show the features extracted by Aroma for all non-keyword tokens.

Table 6.2: Features for selected tokens in Figure 6.4

	Token Feature	Parent Features	Sibling Features	Variable Usage Features
$\texttt{view}_{30}$	#VAR	(#VAR, 2, (#)#) (#VAR, 1, (#)) (#VAR, 1, #.#)	(ViewGroup, #VAR) (#VAR, getChildCount)	((1, #instanceof#), (2, (#)#)) ((2, (#)#), (2, (#)#))
0_{20}	0	(0, 2, #=#) (0, 1, int#) (0, 1, #;#;#;)	(#VAR, 0) (0, #VAR)	-

6.4.3 Recommendation Algorithm

6.4.3.1 Phase I: Light-weight Search

In this phase, Aroma takes a query code snippet, and outputs a list of the top few (e.g. 1000) methods that contain the most overlap with the query. To compute the top methods, we need to compute the degree of overlap between the query and each method body in the corpus. Because our corpus has millions of methods, we need to make sure that the degree of overlap can be computed fast at query time. We use the feature sets of code snippets to compute the degree of overlap, which we call the **overlap score**. Specifically, Aroma intersects the set of features of the query and the method body, and uses the cardinality of the intersection as the overlap score. Computing intersection and its cardinality could be computationally expensive. For efficient computation, we represent the set of features of a code snippet as a sparse vector and perform matrix multiplication to compute the overlap score of all methods with the query code. We next describe the formal details of the phase.

Given a large code corpus containing millions of methods. Aroma parses and creates a simplified parse tree for each method body. It then featurizes each simplified parse tree. Let M be the set of simplified parse trees of all method bodies in the corpus. Aroma also parses

the query code snippet to create its simplified parse tree, say q, and extracts its features. For the simplified parse tree m of each method body in the corpus, we use the cardinality of the set $S(F(m)) \cap S(F(q))$ as an approximate score, called *overlap score*, of how much of the query code snippet overlaps with the method body. Here $S(X)$ denotes the set of elements of the multi-set X, where we ignore the count of each element in the multi-set. Aroma computes a list of η_1 method bodies whose overlap scores are highest with respect to the query code snippet. In our implementation η_1 is usually 1000.

The computation of this list can be reduced to a simple multiplication between a matrix and a sparse vector as follows. The features of a code snippet can be represented as a sparse vector of length $|\mathcal{F}|$—the vector has an entry for each feature in $\mathcal{F}$. If a feature f_i is present in $F(m)$, the multi-set of features of the simplified parse tree m, then the ith entry of the vector is 1 and 0 otherwise. Note that the elements of each vector can be either 0 or 1—we ignore the count of each feature in the vector. To understand this decision, consider a method m that contains a feature f numerous times (say n). Then, say we give Aroma a query q that contains f once. The overlap score between m and q will be increased by n, even though the multiple instances of this feature do not actually indicate greater overlap between m and q. The sparse feature vectors of all method bodies can then be organized as a matrix, say D, of shape $|M| \times |\mathcal{F}|$. Let v_q be the sparse feature vector of the query code snippet q. Then $D \cdot v_q$ is a vector of size $|M|$ that gives the overlap score of each method body with respect to the query snippet. Aroma picks the top η_1 method bodies with the highest overlap scores. Let N_1 be the set of simplified parse trees of the method bodies picked by Aroma.

The corpus we used for evaluation has over 37 million unique features. But each method has an average of 63 methods, so the feature vectors are very sparse. Thus, the matrix multiplication described above can be done efficiently using a fast sparse matrix multiplication library—for our corpus, this phase finishes in less than a second.

6.4.3.2 Phase II: Prune and Rerank

In the following phases, we need a sub-algorithm to compute a maximal code snippet that is common to two given code snippets. For example, given the code snippets `x = 1; y = 2;` and `y = 2; z = 3;`, we need an algorithm that computes `y = 2;` as the intersection of the two code snippets. This algorithm could be implemented using a longest-common subsequence (LCS) [104] computation algorithm on strings by treating the two code snippets as strings. Such an algorithm was used in SNIFF [35] (which performs natural language small code snippet search). However, LCS does not work well for Aroma because often the common parts between two code snippets may not be exactly similar. To illustrate this point, suppose we are given the two code snippets `x = 1; if (y > 1) if (z < 0) w = 4;` and `if (z < 0) if (y > 1) w = 4; v = 10;`, where we have swapped the nesting of the two `if` statements. LCS will retrieve either `if (y > 1) w = 4;` or `if (z < 0) w = 4;` as the intersection, i.e. LCS drops one of the `if` statements along with the non-common assignment statements. Ideally, we should have both `if` statements in the intersection, i.e. the intersection algorithm should compute either `if (y > 1) if (z < 0) w = 4;` or `if (z < 0) if (y > 1) w = 4;` as the intersection.

The example also shows that we can have at most two intersected code snippets when fuzzy similarity exists between the given code snippets—a snippet will either be most similar to the first snippet, or the second snippet. We resolve this ambiguity by picking the intersected snippet that is close to the second snippet. Thus, we can think of the intersection as a code snippet obtained by taking the second snippet, and dropping its fragments which have no resemblance to the first snippet. That is, the algorithm is *pruning* the second snippet while retaining the parts common with the first one.

A simple way to *prune* the second snippet is to look at its parse tree and find a subtree which is most similar to the first snippet. However, such an algorithm will be expensive

because there are exponentially many subtrees in a given tree. Instead, AROMA uses a greedy algorithm which gives us a maximal subtree of the second snippet's parse tree. We have also observed that if we can identify all the leaf nodes in the second snippet's parse tree that need to be present in the intersection, we can get a maximal subtree by simply retaining all the nodes and edges in the tree that lie in a path from the root to the identified leaf nodes. We next formally describe the pruning algorithm.

Let us assume we are given two code snippets, say m_1 and m_2, in the form of their parse trees. The computation of the optimal pruned simplified parse tree, say m_p, requires us to find a subset, say R, of the leaf nodes of m_2. Recall that the set of leaf nodes of m is denoted by $N(m)$ and contains exactly the non-keyword tokens in the parse tree. The set R should be such that the similarity between m_p and m_1 is maximal. We will use the cardinality of the multi-set intersection of the features of two code snippets as their similarity score. That is, the similarity score between two snippets given as parse trees, say m_1 and m_2, is $|F(m_1) \cap F(m_2)|$. Let us denote it by $\mathit{SimScore}(m_1, m_2)$. Once we have computed the set of leaf nodes (i.e. R) that need to be present in the intersection, m_p is the subtree consisting of the nodes in R, and any internal nodes and edges in m_2 which are along a path from any $n \in R$ to the root node in m_2. The greedy algorithm for computing R is described in Algorithm Prune.

Algorithm Prune$(F(m_1), m_2)$:

1. $R \leftarrow \emptyset$.

2. $F \leftarrow \emptyset$.

3. Find n such that

$$n = \text{argmax}_{n' \in N(m_2) - R} \mathit{SimScore}(F(m_1), (F \uplus F(n')))$$

and

$$SimScore(F(m_1), F \uplus F(n)) > SimScore(F(m_1), F).$$

4. If such an n exists, then $R \leftarrow R \cup \{n\}$ and $F \leftarrow F \uplus F(n)$. Go back to Step 3.

5. Else return m_p where m_p is obtained from m by retaining all the non-keyword tokens in R and any internal node or edge which appears in a path from a $n \in R$ to the root of m_2. Then $\texttt{Prune}(F(m_1), m_2)$ is m_p.

In the algorithm, AROMA maintains the collection of the features of the intersected snippet in the variable F. The variable R maintains the set of leaf nodes in the intersected code. Initially, the algorithm starts with an empty set of leaf nodes. It then iteratively adds more leaf nodes to the set from the parse tree of the second method (i.e. m_2). A node n is added if it increases the similarity between the first method and the tree that can be obtained from R. Since F maintains the features of the tree that can be constructed from R, we can get the features of $R \cup \{n\}$ by simply adding the features associated with n (i.e. $F(n)$) to F. If such a node cannot be found, the algorithm constructs the intersected tree from R and returns it.

We are next going to show how AROMA uses the pruning algorithm to rerank the snippets retrieved in Phase 1. Given a query, say q, and a method body, say m, pruning of the method with respect to the query (i.e. $\texttt{Prune}(F(q), m)$) gives a code snippet that is common to both the query and method. If we consider the similarity score between the query and the pruned code snippet, the score should be an alternative way to quantify the overlap between the query and the method. We found empirically that if we use this alternative score to rerank the methods retrieved in Phase 1 (i.e. N_1), then our ranking of search results improves slightly. AROMA uses the reranked list, which we call N_2, in the next phase for clustering and intersection. Note that the pruning algorithm is greedy, so we may not find the best

intersection between two code snippets. In Section 6.6, we show that in very rare cases the greedy pruning algorithm may not give us the best recommended code snippets.

Listing 6.5 shows a code snippet from the reranked search results for the query code snippet in Listing 6.4. In the code snippet, the highlighted tokens are selected by the pruning algorithm to maximize the similarity score to the query snippet.

6.4.3.3 Phase III: Cluster and Intersect

In the final phase, AROMA prepares recommendations by clustering and intersecting the reranked search results from the previous phase. Clustering and intersection are computationally expensive. Therefore, we pick from the list of search results the top $\eta_2 = 100$ methods whose overlap score with the query is above a threshold $\tau_1 = 0.65$, and run the last phase on them. In the discussion below, we assume that N_2 has been modified to contain the top η_2 search results.

Clustering. AROMA clusters together method bodies that are similar to each other. The clustering step is necessary to avoid creating redundant recommendations—for each cluster, only one recommendation is generated. Furthermore, the methods in a cluster may contain unnecessary, extraneous code fragments. An intersection of the code snippets in a cluster helps to create a concise recommendation by getting rid of these unnecessary code fragments.

A cluster contains method bodies that are similar to each other. Specifically, a cluster must satisfy the following two constraints:

1. If we intersect the snippets in a cluster, we should get a code snippet that has more code fragments than the query. This ensures that AROMA's recommendation (which is obtained by intersecting the snippets in the cluster) is an extension to the query

snippet.

2. The pruned code snippets in a cluster are similar to each other. This is because Aroma has been designed to perform search that can tolerate some degree of differences between the query and the results. As such, two code snippets may overlap with different parts of the query. If two such code snippets are part of a cluster, then their intersection will not contain the query snippet. Therefore, the recommendation, which is obtained by intersecting all the snippets in a cluster, will not contain any part of the query. This is undesirable because we want a recommendation that contains the query and some extra new code.

Moreover, Aroma does not require the clusters to be disjoint.

Because of these constraints on a cluster, we cannot simply use a textbook clustering algorithm such as k-means, DBSCAN, or Affinity Propagation. We tried using those clustering algorithms initially (ignoring the constraints) and got poor results. Therefore, we developed a custom clustering algorithm that takes the constraints into account. At a high level, the clustering algorithm starts by treating each method body as a separate cluster. Then, it iteratively merges a cluster with another cluster with single snippet provided that the merged cluster satisfies the cluster constraints and the size of the recommended code snippet from the merged cluster is minimally reduced. We next formally describe the clustering algorithm.

We use $N_2(i)$ to denote the tree at index i in the list N_2. A cluster is a tuple of indices of the form $(i_1, \ldots, i_k)$, where $i_j < i_{j+1}$ for all $1 \leq j < k$. A tuple $(i_1, \ldots, i_k)$ denotes a cluster containing the code snippets $N_2(i_1), \ldots, N_2(i_k)$. We define the commonality score of the tuple $\tau = (i_1, \ldots, i_k)$ as

$$\mathsf{cs}(\tau) = | \cap_{1 \leq j \leq k} F(N_2(i_j))|$$

Similarly, we define the commonality score of the tuple $\tau = (i_1, \ldots, i_k)$ with respect to the query q as

$$\texttt{csq}(\tau) = | \cap_{1 \leq j \leq k} F(\texttt{Prune}(F(q), N_2(i_j)))|$$

We say that a tuple $\tau = (i_1, \ldots, i_k)$ is a *valid tuple* or a *valid cluster* if

1. $\texttt{l}(\tau) = \texttt{cs}(\tau)/\texttt{csq}(\tau)$ is greater than some user-defined threshold τ_2 (which is 1.5 in our experiments). This ensures that after intersecting all the snippets in the cluster, we get a snippet that is at least τ_2 times bigger than the query code snippet.

2. $\texttt{s}(\tau) = \texttt{csq}(\tau)/|F(N_2(i_1))|$ is greater than some user-defined threshold τ_3 (which is 0.9 in our experiments). This requirement ensures that the trees in the cluster are not too similar to each other. Specifically, it says that the intersection of the pruned snippets in a cluster should be very similar to the first pruned snippet.

The set of valid tuples $\mathcal{C}$ is computed iteratively as follows:

1. $\mathcal{C}_1$ is the set $\{(i) \mid 1 \leq i \leq |N_2| \text{ and } (i) \text{ is a valid tuple}\}$.

2. $\mathcal{C}_{\ell+1} = \mathcal{C}_\ell \cup \{(i_1, \ldots, i_k, i) \mid (i_1, \ldots, i_k) \in \mathcal{C}_\ell$ and $i_k < i \leq |N_2|$ and $(i_1, \ldots, i_k, i)$ is a valid tuple and $\forall j$ if $i_k < j \leq |N_2|$ then $\texttt{l}((i_1, \ldots, i_k, i)) \geq \texttt{l}((i_1, \ldots, i_k, j))\}$

Aroma computes $\mathcal{C}_1$, $\mathcal{C}_2$, ...iteratively until it finds an ℓ such that $\mathcal{C}_\ell = \mathcal{C}_{\ell+1}$. $\mathcal{C} = \mathcal{C}_\ell$ is then the set of all clusters. We developed this custom clustering algorithm because existing popular clustering algorithms such as k-means, DBSCAN and Affinity Propagation all gave poor recommendations. Our clustering algorithm makes use of several similarity metrics (the containment score, the Jaccard similarity of various feature sets), whereas standard clustering algorithms usually depend on a single notion of distance. We found the current best similarity metric and clustering algorithm through trial and error.

After computing all valid tuples, Aroma sorts the tuples in ascending order on the first index in each tuple and then in descending order on the length of each tuple. It also drops any tuple τ from the list if it is similar (i.e. has a Jaccard similarity more than 0.5) to any tuple appearing before τ in the sorted list. This ensures that the recommended code snippets are not too similar to each other. Let N_3 be the sorted list of the remaining clusters.

Intersection. Aroma creates a recommendation by intersecting all the snippets in each cluster. The intersection algorithm uses the `Prune` function and ensures that the intersection does not discard any code fragment that is part of the query. Formally, given a tuple $\tau = (i_1, \ldots, i_k)$, `Intersect`$(\tau, q)$ returns a code snippet that is the intersection of the code snippets $N_2(i_1), \ldots, N_2(i_k)$ while ensuring that we retain any code that is similar to q. `Intersect`$((i_1, \ldots, i_k), q)$ is defined recursively as follows:

- `Intersect`$((i_1), q) =$ `Prune`$(F(q), N_2(i_1))$.
- `Intersect`$((i_1, i_2), q) =$ `Prune`$(F(N_2(i_2)) \uplus F(q), N_2(i_1))$.
- `Intersect`$((i_1, \ldots, i_j, i_{j+1}), q) =$ `Prune`$(F(N_2(i_{j+1})) \cup F(q),$ `Intersect`$((i_1, \ldots, i_j), q))$.

In the running example, Listing 6.5 and Listing 6.6 form a cluster. Aroma prunes Listing 6.5 with respect to the union set of features of the query code and Listing 6.6 as the intersection between Listing 6.5 and Listing 6.6. The result of the intersection is shown in Listing 6.7, which is returned as the recommended code snippet from this cluster.

Finally, Aroma picks the top K (where $K = 5$ in our implementation) tuples from N_3 and returns the intersection of each tuple with the query code snippet as recommendations.

```
1 if (!(view instanceof EditText)) {
2     view.setOnTouchListener(new View.OnTouchListener() {
3         public boolean onTouch(View v, MotionEvent event) {
4             hideKeyBoard();
5             return false;
6         }
7     });
```

```
8 }
9 if (view instanceof ViewGroup) {
10    for (int i = 0; i < ((ViewGroup) view).getChildCount(); i++) {
11       View innerView = ((ViewGroup) view).getChildAt(i);
12       setupUIToHideKeyBoardOnTouch(innerView);
13    }
14 }
```

Listing 6.5: A method body containing the query code snippet in Listing 6.4. The highlighted text represents tokens selected in the pruning step.[4]

```
1 if (!(view instanceof EditText)) {
2    view.setOnTouchListener(new View.OnTouchListener() {
3       public boolean onTouch(View v, MotionEvent event) {
4          Utils.toggleSoftKeyBoard(LoginActivity.this, true);
5          return false;
6       }
7    });
8 }
9 if (view instanceof ViewGroup) {
10    for (int i = 0; i < ((ViewGroup) view).getChildCount(); i++) {
11       View innerView = ((ViewGroup) view).getChildAt(i);
12       setupUI(innerView);
13    }
14 }
```

Listing 6.6: Another method containing the query code snippet in Listing 6.4. The highlighted text represents tokens selected in the pruning step.[5]

```
1 if (!(view instanceof EditText)) {
2    view.setOnTouchListener(new View.OnTouchListener() {
3       public boolean onTouch(View v, MotionEvent event) {
4          // your code...
5          return false;
6       }
7    });
8 }
9 if (view instanceof ViewGroup) {
10    for (int i = 0; i < ((ViewGroup) view).getChildCount(); i++) {
11       View innerView = ((ViewGroup) view).getChildAt(i);
12       setupUIToHideKeyBoardOnTouch(innerView);
13    }
14 }
```

Listing 6.7: A recommended code snippet created by intersecting code in Listing 6.5 and Listing 6.6. Extra lines are highlighted.

[4]Adapted from `https://github.com/arcbit/arcbit-android/blob/master/app/src/main/java/com/arcbit/arcbit/ui/SendFragment.java#L468`. Accessed in August 2018.

[5]Adapted from `https://github.com/AppLozic/Applozic-Android-Chat-Sample/blob/master/Applozic-Android-AV-Sample/app/src/main/java/com/applozic/mobicomkit/sample/LoginActivity.java#L171`. Accessed in August 2018.

6.5 Evaluation of Aroma's Code Recommendation Capabilities

Our goal in this section is to assess how Aroma code recommendation can be useful to programmers. To do so, we collected real-world code snippets from Stack Overflow, used them as query snippets, and inspected the code recommendations provided by Aroma to understand how they can add value to programmers in various ways.

6.5.1 Datasets

We instantiated Aroma on 5,417 GitHub projects where Java is the main language and Android is the project topic. We ensured the quality of the corpus by picking projects that are not forked from other projects, and have at least 5 stars. A previous study [75] shows that duplication exists pervasively on GitHub. To make sure Aroma recommendations are created from multiple different code snippets, rather than the same code snippet duplicated in multiple locations, we removed duplicates at project level, file level, and method level. We do this by taking hashes of these entities and by comparing these hashes. After removing duplicates, the corpus contains 2,417,125 methods.

For evaluation, we picked the 500 most popular questions on Stack Overflow with the *android* tag. From these questions, we only considered the top voted answers. From each answer, we extracted all Java code snippets containing at least 3 tokens, a method call, and less than 20 lines, excluding comments. We randomly picked 64 from this set of Java code snippets. We then used these code snippets to carry out the experimental evaluations in the following two sections. In these experiments, we found that on average Aroma takes 1.6 seconds end-to-end to create recommendations on a 24-core CPU. The median response time is 1.3s and 95% queries complete in 4 seconds. A 24-core server was not necessary to achieve reasonable

response time: We reran our experiments on a 4-core desktop machine, and the average response time is 2.9 seconds. We believe this makes Aroma suitable for integration into the development environment as a code recommendation tool.

6.5.2 Recommendation Performance on Partial Code Snippets

In this experiment, we manually created partial code snippets by taking the first half of the statements from each of the 64 code snippets. Since each full code snippet from Stack Overflow represents a popular coding pattern, we wanted to check whether Aroma could recommend the missing statements in the code snippet given the partial query code snippet. We always selected the first half of each code snippet to avoid subjective bias. Since we know how the tool works, we would be inclined to pick the lines that we think will produce better results. On average, the query code snippets were 1 to 5 lines and contained 10 to 100 features.

We could not extract partial query code snippets from 14 out of 64 code snippets because they contained a single statement. Single-statement snippets do get recommendations, but since we do not have a ground truth, we cannot judge their quality objectively. For the remaining 50 query code snippets, Aroma recommendations fall into the following two categories.

6.5.2.1 Exact Recommendations.

In 37 cases (74%), one of the top 5 Aroma recommendations matched the original code snippet. Example D in Table 6.1 shows a partial query snippet which included the first two statements in a try-catch block of a Stack Overflow code snippet, and Aroma recommended the same error handling code as in the original code snippet.

6.5.2.2 Alternative Recommendations.

In the other 13 cases (26%), none of the Aroma recommended code snippets matched the original snippets. While in each case the recommended snippets did not contain the original usage pattern, they still fall in some of the categories in Table 6.3 which we discuss in the next section. Example E in Table 6.1 shows a partial code snippet which included one of two common ways to send an object with an `Intent`. Given the statement, Aroma did not recommend the other way to serialize an object in the original code snippet, but suggested a customary way to start an activity with an `Intent` containing a serialized object.

6.5.3 Recommendation Quality on Full Code Snippets

In this experiment, we used the 64 code snippets as queries to evaluate the quality of Aroma's recommendations. While the experiment in the previous section used *partial* snippets extracted from each of the 64 code snippets, here we used the *full* code snippets. This meant that we could use all 64 snippets instead of just the 50 used in Section 6.5.2, as we did not have to filter out single-statement code snippets.

We manually inspected the recommended code snippets and determined whether they are useful. We considered a recommended code snippet to be "useful" if in a programming scenario where a programmer writes the query code, they would benefit from seeing the related methods or common usage patterns in the code recommendations. We classified the recommended snippets into several categories by how the recommended code relates to the query snippet. The classification is subjective because there is no "ground truth" on what the recommended code should be, and the actual usefulness depends on how familiar the programmer is with the language and framework. Nevertheless, we present the categories and some examples in Table 6.1 to demonstrate the variety of code recommendations Aroma can provide. Two of the authors did the manual inspection and categorization, and two other

authors verified the results.

6.5.3.1 Configuring Objects

In this category, the recommended code suggests additional configurations on objects that are already appearing in the query code. Examples include adding callback handlers, and setting additional flags and properties of an existing object. Listings 6.1, 6.2 in the introduction, as well as Example A in Table 6.1 shows examples of this category. These recommendations can be helpful to programmers who are unfamiliar with the idiomatic usages of library methods.

6.5.3.2 Error Checking and Handling

In this category, the recommended code adds null checks and other checks before using an object, or adds a try-catch block that guards the original code snippet. Such additional statements are useful reminders to programmers that the program might enter an erroneous state or even crash at runtime if exceptions and corner cases are not carefully handled. Listings 6.1, 6.3 in the introduction show an example of this category.

6.5.3.3 Post-processing

The recommended code extends the query code to perform some common operations on the objects or values computed by the query code. For example, recommended code can show API methods that are commonly called. Example B in Table 6.1 shows an example of this category, where the recommendation applies Gaussian blurring on the decoded bitmap image. This pattern is not obligatory but demonstrates a possible effect that can be applied on the original object. This category of recommendations can help programmers discover related methods for achieving certain tasks.

6.5.3.4 Correlated Statements

The recommended code adds statements that do not affect the original functionalities of the query code, but rather suggests related statements that commonly appear alongside the query code. In Example C in Table 6.1, the original code moves the cursor to the end of text in an editable text area, where the recommended code also configures the Android Support Action Bar to show the home button and hide the activity title in order to create a more focused view. These statements are not directly related to the text view, but are common in real-world code.

6.5.3.5 Unclustered Recommendations

In rare cases, the query code snippet could match method bodies that are mostly different from each other. This results in clusters of size 1. In these cases, Aroma performs no intersection and recommends the full method bodies without any pruning.

The number of recommended code snippets for each category is listed in Table 6.3. For recommendations that belong to multiple categories, we counted them for each of the categories. We believed the first four categories all can be useful to programmers in different ways, where the unclustered recommendations may not be. For 59 out of the 64 query code snippets (92%), Aroma generated at least one useful recommended snippet that falls in the first four categories.

Table 6.3: Categories of Aroma code recommendations

Category	Count
Configuring Objects	17
Error Checking and Handling	14
Post-processing	16
Correlated Statements	21
Unclustered Recommendations	5

6.5.4 Comparison with Pattern-Oriented Code Completion

Pattern-oriented code completion tools [87, 91, 93] could also be used for code recommendation. For example, GraPacc [91] proposed using mined API usage patterns for code completion. To compare GraPacc's code recommendation capabilities to Aroma's, we took the dataset of 15 Android API usage patterns manually curated from Stack Overflow posts and Android documentation by the authors of BigGroum [87]. We used BigGroum's dataset because this tool extends the pattern-mining tool Groum to scale to large corpora with over 1000 repos. While there are more recent ML-based code completion tools, they focus on completing the next token or predicting the correct API method to invoke, which does not directly compare to Aroma. Among the 15 snippets in this dataset, 11 were found in BigGroum mining results. Therefore, if GraPacc is instantiated on the patterns mined by BigGroum, 11 out of the 15 patterns could be recommended by GraPacc.

In order to evaluate Aroma, we followed the same methodology as in Section 6.5.2 to create a partial query snippet from each of the 15 full patterns, and checked if any of the Aroma recommended code snippets contained the full pattern. For 14 out of 15 patterns, Aroma recommended code containing the original usage patterns, i.e. they are *exact recommendations* as defined in Section 6.5.2.1. An advantage of Aroma is that it could recommend code snippets that do not correspond to any previously mined pattern by BigGroum. Moreover, Aroma could recommend code which may not contain any API usage.

6.6 Evaluation of Search Recall

One of the most important and novel phases of the Aroma's code recommendation algorithm is phase II: prune and rerank, which produces the *reranked search results*. The purpose of this phase is to rank the search results from phase I (i.e. the light-weight search phase) so

that any method containing most parts of the query code is ranked higher than a method body containing a smaller part of the query code. Therefore, if a method contains the entire query code snippet, it should be ranked top in the reranked search result list. However, in rare cases this property of Aroma may not hold due to two reasons: 1) Aroma's pruning algorithm is greedy and approximate due to efficiency reasons, and 2) the kinds of features that we extract may not be sufficient.

To evaluate the recall of the prune and rerank phase, we created a micro-benchmark dataset by extracting partial query code snippets from existing method bodies in the corpus. On each of these query snippets, Aroma should rank the original method body as number 1 in the reranked search result list, or the original method body should be 100% similar to the first code snippet in the ranked results. We created two kinds of query code snippets for this micro-benchmark:

- *Contiguous code snippets.* We randomly sampled 1000 method bodies with at least 12 lines of code. From each method body we take the first 5 lines to form a partial query code snippet.
- *Non-contiguous code snippets.* We again randomly sampled 1000 method bodies with at least 12 lines of code. From each method body we randomly sample 5 lines to form a partial query code snippet.

We first evaluated Aroma's search recall on this dataset. We employed statistical bootstrapping to minimize sampling bias from the dataset. Then, we compared it with alternative setups using clone detectors and conventional search techniques. The results are reported in Table 6.4.

Table 6.4: Comparison of recall between a clone detector, conventional search techniques, and AROMA

	Contiguous		Non-contiguous	
	Recall@1	Recall@100	Recall@1	Recall@100
SCC	(12.2%)		(7.7%)	
Keywords Search	78.3%	96.9%	93.0%	99.9%
Features Search	78.3%	96.8%	88.1%	98.6%
AROMA	**99.1%**	**100%**	**98.3%**	**100%**

6.6.1 Comparison with Clone Detectors and Conventional Search Techniques

AROMA's search and pruning phases are somewhat related to clone detection and conventional code search. In principle, AROMA can use a clone detector or a conventional code search technique to first retrieve a list of methods that contain the query code snippet, and then cluster and intersect the methods to get recommendations. We tested these alternative setups for search recall on the same micro-benchmark dataset.

6.6.1.1 Clone Detectors

SOURCERERCC [116] is a state-of-the-art clone detector that supports Type-3 clone detection. We wanted to compare AROMA with SOURCERERCC to examine whether a current-generation clone detector can be used as the light-weight search phase in AROMA.

We instantiated SOURCERERCC on the same corpus indexed by AROMA. We then used SOURCERERCC to find clones of the same 1000 contiguous and non-contiguous queries in the micro-benchmark suite. SOURCERERCC retrieved all similar methods above a certain similarity threshold, which is 0.7 by default. However, it does not provide any similarity score between two code snippets, so we were unable to rank the retrieved results and report recall at a specific ranking. We could modify SOURCERERCC to return the similarity scores,

but we do not expect the results to change.

SourcererCC's recall was 12.2% and 7.7% for contiguous and non-contiguous code queries, respectively. SourcererCC indexes at method-level granularity, and only returns methods whose entire body matches the query code. We also found that in many cases SourcererCC found code snippets *shorter* than the query snippet. While these are Type-3 clones by definition, they are not useful for generating code recommendations in Aroma. Extending SourcererCC to return the methods enclosing the clone snippets found would not work, because it does not consider the methods enclosing the target snippets as "clones" in the first place. We worked closely with a member of the SourcererCC team, and found that making SourcererCC find all occurrences of an arbitrary code snippet, contiguous and non-contiguous, would require significant reengineering. Therefore, we conclude that current-generation clone detectors may not suit Aroma's requirements for light-weight search.

6.6.1.2 Conventional Search Using TF-IDF and Structural Features

We implemented a conventional code search technique using classic TF-IDF [118]. Specifically, instead of creating a binary vector in the featurization stage, we created a normalized TF-IDF vector. We then created the sparse index matrix by combining the sparse vectors for every method body. The $(i, j)^{\text{th}}$ entry in the matrix is defined as:

$$tfidf(i,j) = (1 + \log tf(i,j)) \cdot \log \frac{J}{df(i)}$$

where $tf(i,j)$ is the count of occurrences of feature i in method j, and $df(i)$ is the number of methods in which feature i exists. J is the total number of methods. During retrieval, we created a normalized TF-IDF sparse vector from the query code snippet, and then took its dot product with the feature matrix. Since all vectors are normalized, the result contains

the cosine similarity between the feature vectors of the query and of every method. We then returned the list of methods ranked by their cosine similarities.

6.6.1.3 Conventional Search Using TF-IDF and Keywords

We implemented another conventional code search technique by simply treating a method body as a bag of words and using the standard TF-IDF technique for retrieval. To do so, we extracted words instead of structural features from each token, and used the same vectorization technique as in Section 6.6.1.2.

As shown in Table 6.4, the recall rates of both conventional search techniques are considerably lower than Aroma. We observed that in many cases, though the original method was present in the top 100 results, it was not the top result because there are other methods with higher similarity scores due to more overlapping features or keywords. Without pruning, there is no way to determine how well a method contains the query code snippet. This experiment shows that pruning is essential in order to create a precise ranked list of search results.

6.7 Conclusion

We presented Aroma, a new tool for code recommendation via structural code search. Aroma works by first indexing a large code corpus. It takes a code snippet as input, assembles a list of method bodies from the corpus that contain the snippet, and clusters and intersects those method bodies to offer several succinct code recommendations.

To evaluate Aroma, we indexed a code corpus with over 2 million Java methods, and performed Aroma searches with code snippets chosen from the 500 most popular Stack Overflow questions with the *android* tag. We observed that Aroma provided useful recom-

mendations for a majority of these snippets. Moreover, when we used half of the snippet as the query, AROMA exactly recommended the second half of the code snippet in 37 out of 50 cases.

Further, we performed a large-scale automated evaluation to test the accuracy of AROMA search results. We extracted partial code snippets from existing method bodies in the corpus and performed AROMA searches with those snippets as the queries. We found that for 99.1% of contiguous queries and 98.3% of non-contiguous queries, AROMA retrieved the original method as the top-ranked result. We also showed that AROMA's search and pruning algorithms are decidedly better at finding methods *containing* a code snippet than conventional code search techniques.

Our ongoing work shows that AROMA has the potential to be a powerful developer tool. Though new code is frequently similar to existing code in a repository, currently available code search tools do not leverage this similar code to help programmers add to or improve their code. AROMA addresses this problem by identifying common additions or modifications to an input code snippet and presenting them to the programmer in a concise, convenient way.

Chapter 7

Method-level Recommendation for Related Code

7.1 Introduction

Over the past decade, code search has emerged as an interesting, but challenging, topic to both industry and research communities. Various code search techniques have been proposed in the literature [22, 74, 79, 107], and some code search engines have been implemented and are, or were, publicly available [1, 2, 3, 4, 5, 73]. Code search engines take some specification as input (e.g., a keyword description, a code fragment, or a test) and recommend pieces of code that match the given specification based on some forms of "similarity" measurement. Prior work has shown that by identifying and contrasting similar code, programmers could quickly understand the gist of implementing a function and explore potential variations to write more complete and robust code [76, 153]. For instance, by inspecting variations in similar code found in GitHub, programmers are able to identify critical code parts such as safety checks and exception handling logics that are missed in a given code example [153].

This work explores the opportunities of searching relevant code beyond similarity. Consider the following scenario. A programmer is implementing a Java method for file decompression. A code search engine may recommend a code example as as shown in Listing 7.1 in Figure 7.1, which takes the path to a zip file as input and unpacks all files within the zip into a target directory. This piece of code is sufficient for a simple program task of unpacking a zip file. However, in practice, the programmer may undertake a more complex programming task where unzipping a file is a small, integral part. Therefore, the programmer may also want to know what else may be related to this functionality. If an additional functionality often co-occurs with unzipping, the programmer may want to add it to her own project as needed. For instance, Listing 7.2 shows an example of this kind of additional functionalities—a method that zips a list of files from a folder into the target zip file. Unzipping and zipping are two kinds of file manipulation in the opposite direction. Though these two functions work independently, they are often implemented together in a code base to complement each other. We consider the zip method and the unzip method *related* to each other, or *complementary code fragment*, to be more specifically.

Code-to-code search engines could be leveraged to identify related code given a code fragment of interest [5, 68, 73]. However, these techniques find syntactically or semantically similar code fragments only, without considering about auxiliary or complementary functionality. For instance, given an unzip function, they cannot find a complementary zip function, since neither the implementation nor the functionality of these two operations are similar. Currently, there is also limited understanding about what other kinds of relevant code beyond similar code may exist in practice and thus should be recommended modern code search engines.

Complementary methods shown in Figure 7.1 often co-occur in the same source file or the same code base, which serves as an interesting property to exploit for recommending relevant code examples. Manually identifying co-occurring code is tedious and time-consuming, since

```java
1  public static boolean unpackZip(String path,
       String zipname, String targetDirectory) {
2  try {
3  InputStream is = new FileInputStream(path +
       zipname);
4  ZipInputStream zis = new ZipInputStream(new
       BufferedInputStream(is));
5  byte[] buffer = new byte[1024];
6  int count;
7  while ((ZipEntry ze = zis.getNextEntry()) !=
       null) {
8  String filename = ze.getName();
9  if (ze.isDirectory()) {
10 File fmd = new File(targetDirectory + filename);
11 fmd.mkdirs();
12 continue;
13 }
14 FileOutputStream fout = new
       FileOutputStream(targetDirectory +
       filename);
15 while ((count = zis.read(buffer)) != -1) {
16 fout.write(buffer, 0, count);
17 }
18 fout.close();
19 zis.closeEntry();
20 }
21 zis.close();
22 } catch (IOException e) {
23 e.printStackTrace();
24 return false;
25 }
26 return true;
27 }
```

Listing 7.1: Query code: unpacking a zip file

```java
1  public static void zip(String baseFolder,
       List<File> files, String zipFile) {
2  try {
3  BufferedInputStream origin = null;
4  FileOutputStream dest = new
       FileOutputStream(zipFile);
5  ZipOutputStream out = new ZipOutputStream(new
       BufferedOutputStream(dest));
6  byte data[] = new byte[BUFFER];
7  for (File file : files) {
8  FileInputStream fi = new FileInputStream(file);
9  origin = new BufferedInputStream(fi, BUFFER);
10 String relativeFileName =
       file.getAbsolutePath().replace(baseFolder
       + File.separator , """");
11 ZipEntry entry = new ZipEntry(relativeFileName);
12 out.putNextEntry(entry);
13 int count;
14 while ((count = origin.read(data, 0, BUFFER))
       != -1) {
15 out.write(data, 0, count);
16 }
17 origin.close();
18 }
19 out.close();
20 } catch(Exception e) {
21 e.printStackTrace();
22 }
23 }
```

Listing 7.2: Related code: zipping a file

Figure 7.1: An example of recommending relevant code that complements desired functionality

some co-occurring code may be project-specific and thus not relevant to a given code query. Therefore, we first build an automated approach to identify, cluster, and rank common, co-occurring code given a code query. Our approach uses a state-of-the-art clone detector called SourcererCC [116] to identify similar counterparts (i.e., clones) in a large code corpus. Then our approach contrasts surrounding code of those clones and identify common code that are shared around multiple clones.

Using 21K Java code snippets from Stack Overflow as code queries, we automatically identify relevant code of these code queries in a large corpus of 50K GitHub projects with at least five stars. As a result, we obtained 21K groups of similar code in GitHub. We manually

inspected a random sample of 50 common, co-occurring code fragments and examined their relevancy to the original query. 74% of those common, co-occurring code fragments represented relevant functionality, which should be included in code search results. Furthermore, we identified three major types of relevant co-occurring code—*complementary*, *supplementary*, and *alternative* functions. These findings show that it is beneficial to recommend common, co-occurring code of a given code query to achieve more complete functionalities, instead of just recommending similar code.

To further demonstrate this idea, we implement a Chrome extension called CodeAid that recommends related but non-similar code when programmers browse code examples in Stack Overflow. It is well known that programmers often search and reuse online examples during modern software development [28, 52, 136]. CodeAid augments this programming workflow by reminding programmers what other complementary, supplementary, or alternative functions should also be included as they copy and paste code from the Web. We compared CodeAid with a state-of-the-art code search engine called Facoy [68]. Among ten sample search queries, FaCoy only identified related code for one query since the related code is similar to the code query. A general-purpose search engine, Google Search, was able to identify GitHub files that contain related code recommended by CodeAid for half of the queries. However, Google could only retrieve full files where programmers still had to manually go through those files to identify related code. By contrast, CodeAid pinpointed where related code fragments were based on how frequently they occurred in other similar locations.

In summary, this paper makes the following contribution:

- We present a new code search method that recommends common, co-occurring code of a given code query, rather than only recommending similar code.
- We empirically show the prevalence of common, co-occurring code by quantifying the commonality of surrounding code of GitHub clones. We also find that the majority of such

co-occurring code fragments represent meaningful functionality such as complementary, supplementary, or alternative functions, which should be recommended by modern code search engines.

- We develop a Chrome extension called CodeAid to recommend related code during online code search and demonstrate that CodeAid is capable of recommending related code that cannot be identified by a state-of-the-art code search tool.

The rest of the paper is organized as follows: Section 7.3 describes the approach for generating common, co-occurring methods, with ranking. Section 7.5 presents the manual analysis result of common, co-occurring code in terms of its relevance to the original query. Section7.6 illustrates the Chrome extension, CodeAid, as well as a use scenario of it, and Section 7.7 explores whether recommended code from existing search engines can provide same relevance as results from CodeAid. Section 7.2, 7.8, and 7.9 summarize the related work, point out the threats to validity, and conclude the paper.

7.2 Related Work

Various code search techniques have been proposed to discover relevant code components (e.g., functions, code snippets) given a user query. For example, given a keyword query, Portfolio retrieves function definitions and their usages using a combination of a PageRank model and an association model [80]. Chan et al. improve Portfolio by matching the textual similarity between containing nodes in an API usage subgraph with a keyword query [34]. CodeHow also finds code snippets relevant to a natural language query. It explores API documents to identify relationships between query terms and APIs [77]. Instead of using natural language queries, several techniques automatically recommend relevant code snippets based on contextual information such as types in a target program [62, 102**?** **?**]. CodeGenie

is a test-driven code search technique that allows developers to specify desired functionality via test cases and then matches relevant methods and classes with the given test [74]. To more precisely capture search intent, S6 allows developers to express desired functionality using a combination of input-output types, test cases, and keyword descriptions [107].

Code-to-code search tools are most related to our technique among all different kinds of code search techniques. Given a code snippet as input, FaCoY [68] finds semantically similar code snippets in a Stack Overflow dataset by first matching with accompanied natural language descriptions in related posts instead of matching code directly. Unlike FaCoY, several techniques infer an underlying code search pattern from a given code fragment [81, 82, 121, 151]. Sydit generalizes concrete identifiers (e.g., variable names, types, and method calls) in a given code example as an abstract code template and identifies other similar locations via AST-based tree matching [81]. Lase uses multiple code examples instead of a single example to better infer the search intent of a user [82]. Critics allows developers to construct an AST-based search pattern from a single example through manual code selection, customization, and parameterization [151]. These code-to-code search tools focus on identifying relevant code snippets that are syntactically or semantically similar to a given code snippet. However, since many programming tasks (e.g., password encryption and decryption) require multiple code snippets or functions to work together, none of the existing techniques recommend code snippets that complement a given code snippet to complete desired functionality.

7.3 Data Collection Approach

Our approach takes a code fragment as input and searches a code corpus to identify related code fragments. Given a user-selected code fragment, we first detect its similar methods in the corpus based on syntactic similarity. Then we trace back to the containing files of these similar methods and identifies other co-occurring methods in these files. Among these

co-occurring methods, we further measure each method's similarity to methods in other files, and cluster similar methods. Then we rank co-occurring methods based on the size of cluster it centers. Figure 7.2 describes the pipeline of finding common, co-occurring code fragments given an input code fragment.

7.3.1 Retrieve similar methods

7.3.1.1 Parse a code corpus

We focus on method-level code fragments written in Java in this work. We parse all Java source files to abstract syntax trees (ASTs) and traverse the ASTs to extract all defined methods. Note that the approach is not limited to any programming language. We can switch to any other language by using its particular parser. We use the phrases code fragments and methods interchangeably in the paper.

7.3.1.2 Tokenization

Tokenization is the process of transforming source code into a bag of words. Tokenization starts from removing comments, spaces, tabs and other special characters. Then it identifies distinct tokens and count their frequencies. For each method, the result of tokenization is formatted as a list of tuples such as `(token, freq)`, where the first element is a token in the method and the second element refers to the token occurrence in the method.

We tokenize both the input code fragment and all methods in the code corpus, in preparation for the next step of finding similar pairs.

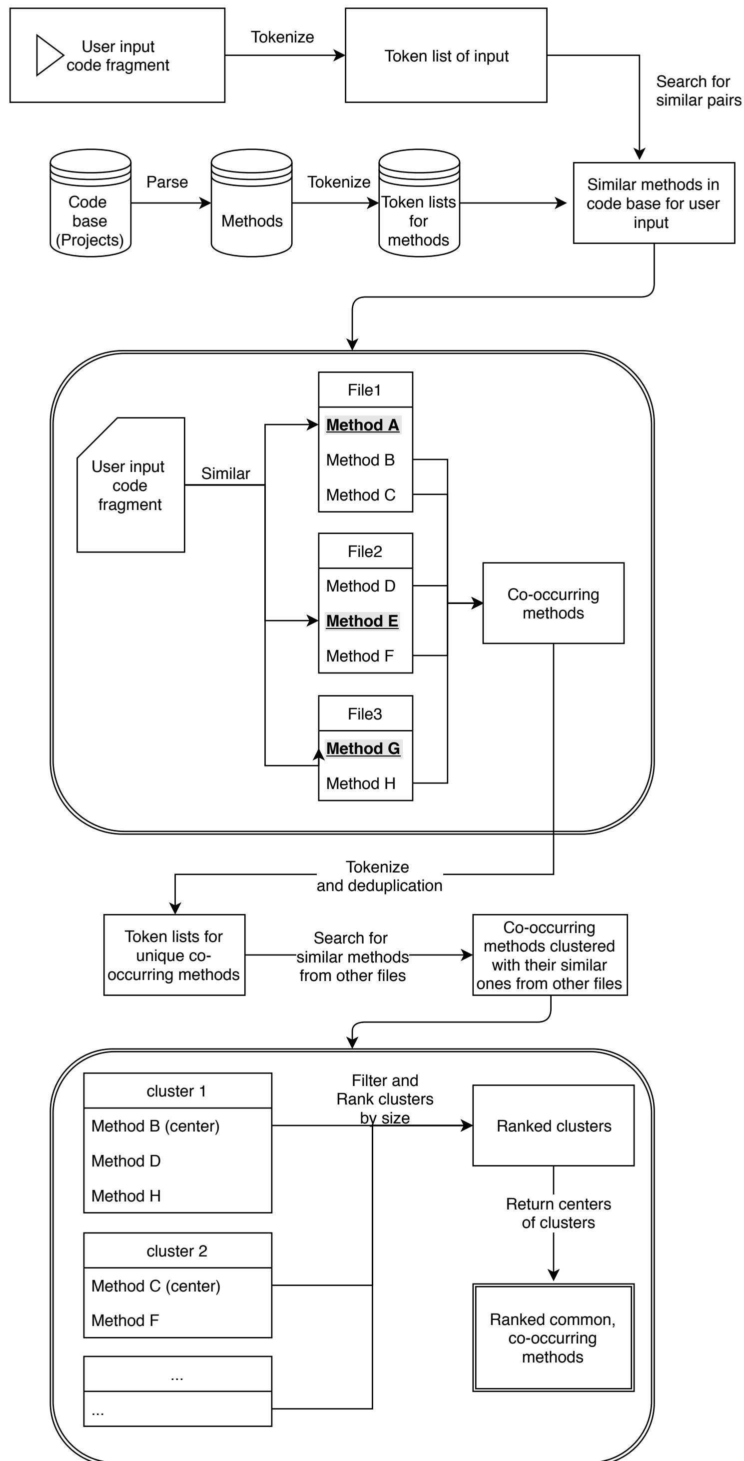


Figure 7.2: The pipeline of collecting common co-occurring methods

7.3.1.3 Search for similar methods

For the input code fragment, we retrieve its similar counterparts from the code corpus using a token-based clone detection tool called SourcererCC [116]. By evaluating the scalability, execution time, recall and precision of SourcererCC, and comparing it to publicly available and state-of-the-art tools, SourcererCC has been shown to have both high recall and precision, and is able to scale to a large repository using a standard workstation. All of the above make SourcererCC a good candidate for retrieving similar counterparts for our inputs.

We use 70% similarity threshold, because it yields the best precision and recall on multiple clone benchmarks [116]. SourcererCC takes the token lists of the input code fragment and all methods in the code corpus, and returns the similar methods to the input in the code corpus. As shown in Figure 7.2, the user input has three similar counterparts in our code corpus, which are `Method A` in `File1`, `Method E` in `File2`, and `Method G` in `File3`.

7.3.2 Identify Co-occurring Code Fragments

Given those similar code fragments identified in the previous step, we trace back to the files that contain these similar counterparts and identify co-occurring methods in the same file as potentially related code fragments. Algorithm 1 gives a more formal description of the process.

`Method A, E, G` are the three similar methods detected by SourcererCC. `File 1, 2, 3` are the three GitHub files contain these similar methods respectively. `File1` also contains `Method B, C`, `File2` has another two methods `Method D, F`, and `Method H` is in `File3`. Therefore, `Method B, C, D, F, H` will be returned as co-occurring methods by Algorithm 1.

Algorithm 1: Identify co-occurring code fragments

```
Data: similar methods
Result: co-occurring methods
initialize resultList;
for m_s in similarMethods do
    ghFiles = traceGitHubFiles(method) ;
    for f in ghFiles do
        methodsInFile = parse(f);
        for m_f in methodsInFile do
            if m_f is not m_s then
                resultList.add(m_f) ;
            end
        end
    end
end
```

7.3.3 Clustering and Ranking

7.3.3.1 Cluster co-occurring code fragments

We further get the token lists for those co-occurring methods identified in the previous step and remove duplicate methods. In order to detect common co-occurring code fragments, we cluster the remaining unique co-occurring methods based on their token similarity. Given each method, we compute its similarity to other methods from different GitHub files. Each method will serve as the center of a cluster, we browse among other methods from different files and add similar methods to the current cluster. Algorithm 2 describes the process.

For the co-occurring methods pool, `Method B, C, D, F, H`, each method will be the center of a cluster. For `Method B`, we compute token similarity with `Method D, F, H` and get two similar ones, `Method D, H`, so we add these two similar methods to the cluster, resulting in cluster size being three. Similarly, we add `Method F` to the cluster centered by `Method C` and get a cluster with size two.

Algorithm 2: Clustering co-occurring code fragments

```
Data: n co-occurring methods
Result: clustered co-occurring methods
initialize clusters = {X_1, X_2, ..., X_n};
for m_i in cooccurringMethods do
    X_i.add(m_i) ;
    for m_j in cooccurringMethods do
        if m_i and m_j do not come from the same file then
            if tokenSimilarity(m_i, m_j) > 0.7) then
                X_i.add(m_j);
            end
        end
    end
end
```

7.3.3.2 Screen and rank clusters by size

After getting the candidate clusters, we keep only clusters with size being at least two. This means the center of the cluster has occurred at least twice among the GitHub files. We rank the remaining clusters by size and return the cluster centers as our final list of common, co-occurring code fragments with ranking. If two clusters have the same size, we will further rank them by the line number distance between the cluster center and the original counterpart of the user input (e.g. `Method C, A`), in ascending order. In our example, we will return `Method B` first, and then `Method C`, as our common co-occurring code fragments.

7.4 Dataset

We apply our approach to Stack Overflow (SO) and GitHub. We use code snippets in SO as the pool of user-input queries and use Java projects in GitHub as our code corpus to search from. We choose these two datasets not only because of their popularity within the programming community, but also because they are part of a larger system of software production. The same users that rely on the hosting and management characteristics of GitHub

often have difficulties and need help on the implementation of their computer programs, seek support on SO for their specific problems, or hints of solutions from ones with a degree of similarity, and return to GitHub to apply the knowledge acquired.

Previous work have shown that developers often copy and paste code snippets from Stack Overflow to their GitHub projects and make adaptations as needed [18, 148, 150, 153]. Our approach will facilitate such opportunistic code reuse process when developers browse code snippets in SO. The use scenario will be: when a user is interested in a code snippet in SO, we recommend related code fragments from GitHub, showing what other code they may also want to investigate and integrate into their own project.

We downloaded Java projects on GitHub by querying GHTorrent [57]. GHTorrent is a scalable, offline mirror of data offered through the GitHub REST API, available to the research community as a service. It provides access to all the metadata of GitHub projects, e.g., the clone url, the number of stars and committers, main programming languages in a project, etc. We use these metadata to screen the projects. Our project selection criteria are:

- We only consider GitHub projects that have at least five stars, in order to avoid toy projects that do not adequately reflect software engineering practices [65].
- We only keep non-forked projects, because project forking leads to many identical projects and would unnecessarily skew our recommendation.
- Prior work on GitHub cloning finds many identical files among GitHub projects, since developers may copy the whole file into another project without making any changes [75]. To account for this internal duplication in GitHub, we remove duplicated GitHub files using the same file hashing method as in [75].

As a result, we downloaded 50,826 non-forked Java repositories with at least five stars from

GitHub. After de-duplication, 5,825,727 distinct Java files remain.

We downloaded the SO dump taken in October 2016 [17]. From the data dump, we extract code snippets in the markdown `<code>` from SO posts with `java` or `android` tags. We consider code snippets in answer posts only, since snippets in question posts are rarely used as valid code examples. This results in 312,219 Java and Android answer posts.

Since SO snippets are often free-standing statements with low parsable rates [149], we used a customized pre-processor before tokenization. For free-standing statements, we wraps them with dummy class and method definitions, and add semicolons after statements as needed. For snippets contain multiple methods, we chunk them into individual ones. We keep only parsable SO snippets after pre-processing.

Prior work finds that larger SO snippets have more meaningful clones in GitHub [150]. Hence, we choose to study SO snippets with no less than 50 tokens after tokenization. We also remove duplicated examples within SO. As a result, we collect 186,392 distinct SO snippets.

We run SoucererCC to find all similar pairs between SO and GitHub. We run on a server machine with 116 cores and 256G RAM. It takes 24 hours to complete. As a result, we get 21,207 distinct SO methods that have one or more similar code fragments in GitHub. The SO snippets have a median of two GitHub clones and a mean of one GitHub clones. The distribution of number of GitHub clones is shown in Figure 7.3. From the distribution we can see that SO snippets most commonly have zero to five similar counterparts in GitHub. Most of the SO snippets have less than twenty GitHub clones.

We collect the original GitHub files which contain the similar counterparts, then we extract all co-occurred methods from these GitHub files. For each co-occurring method in each GitHub file, we cluster its similar counterparts from other files. We keep only the clusters with size of at least two and return the remaining clusters in descending order of size.

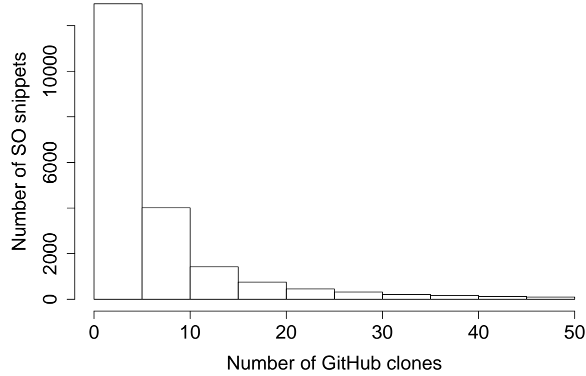


Figure 7.3: Distribution of number of GitHub clones

For 11,110 out of 21K SO queries, we can retrieve common, co-occurring code fragments from GitHub. That is, using our SO query code base and GitHub search code corpus, we can find common co-occurring code for 52.4% of the queries. The SO queries have a median of 24 common co-occurring methods in GitHub and a mean of 74. The retrieved methods have a median of 12 average lines of code, and a mean of 14 average lines of code. The distribution of number of retrieved methods and distribution of average lines of code are shown in Figure 7.4 and 7.5 respectively. From the figures we can see that a SO query is most likely to have less than ten common co-occurring code fragments, and most of the SO queries have less than 50 common co-occurring methods in GitHub. Most retrieved methods for a query will have five to thirty average lines of code.

7.5 Manual Analysis and Categorization

We randomly select 50 SO snippets with its common co-occurring code fragments from the 11,110 groups, and manually examine whether these code fragments are related to the SO input or not, and categorize why we call the relationship a relevant one.

We use *Precision@k* metric to evaluate the common co-occurring code which is defined as

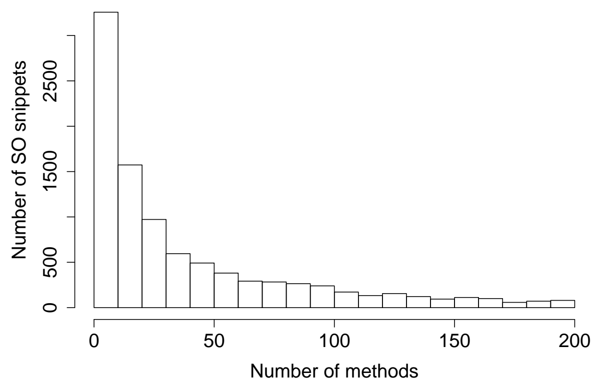


Figure 7.4: Distribution of number of common co-occurring methods

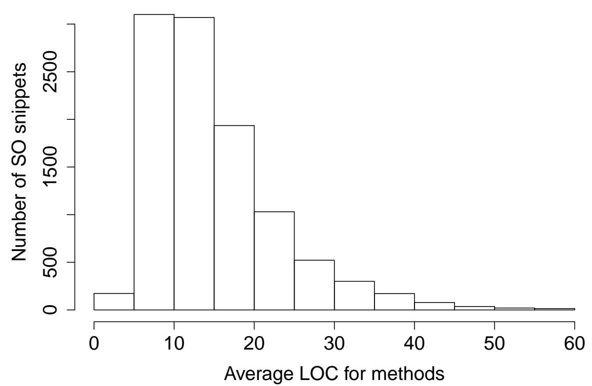


Figure 7.5: Distribution of average LOC of common co-occurring methods

Table 7.1: Complementary method examples

Query Code Snippet	Related Code

<pre>1 public static byte[] encrypt(final SecretKeySpec
      key, final byte[] iv, final byte[] message)
2 throws GeneralSecurityException {
3   final Cipher cipher =
        Cipher.getInstance(AES_MODE);
4   IvParameterSpec ivSpec = new
        IvParameterSpec(iv);
5   cipher.init(Cipher.ENCRYPT_MODE, key, ivSpec);
6   byte[] cipherText = cipher.doFinal(message);
7   log(""cipherText"", cipherText);
8   return cipherText;
9 }</pre>
	<pre>1 public static byte[] decrypt(final SecretKeySpec
	      key, final byte[] iv, final byte[]
	      decodedCipherText)
	2 throws GeneralSecurityException {
	3 final Cipher cipher =
	        Cipher.getInstance(AES_MODE);
	4 IvParameterSpec ivSpec = new
	        IvParameterSpec(iv);
	5 cipher.init(Cipher.DECRYPT_MODE, key, ivSpec);
	6 byte[] decryptedBytes =
	        cipher.doFinal(decodedCipherText);
	7 log(""decryptedBytes"", decryptedBytes);
	8 return decryptedBytes;
	9 }</pre>
	Example B: Complementary method
	• The query snippet implements `encrypt` functionality for an byte array.
	• The related method decrypts a decoded byte array.

<pre>1 public void onCreate(Bundle savedInstanceState) {
2   super.onCreate(savedInstanceState);
3   setContentView(R.layout.main);
4   preferred =
        (TextView)findViewById(R.id.preferred);
5   orientation =
        (TextView)findViewById(R.id.orientation);
6   mgr = (SensorManager)
        this.getSystemService(SENSOR_SERVICE);
7   accel =
        mgr.getDefaultSensor(Sensor.TYPE_ACCELEROMETER);
8   compass =
        mgr.getDefaultSensor(Sensor.TYPE_MAGNETIC_FIELD);
9   orient =
        mgr.getDefaultSensor(Sensor.TYPE_ORIENTATION);
10 WindowManager window = (WindowManager)
11 this.getSystemService(WINDOW_SERVICE);
12 int apiLevel =
        Integer.parseInt(Build.VERSION.SDK);
13 if(apiLevel <8) {
14   mRotation =
          window.getDefaultDisplay().getOrientation();
15 }
16 else {
17   mRotation =
          window.getDefaultDisplay().getRotation();
18 }
19 }</pre>
	<pre>1 protected void onPause() {
	2 mgr.unregisterListener(this, accel);
	3 mgr.unregisterListener(this, compass);
	4 mgr.unregisterListener(this, orient);
	5 super.onPause();
	6 }</pre>
	Example C: Complementary method
	• The query snippet implements `onCreate` functionality for an `Android` app activity.
	• The related method implements `onPause` which does not have direct function call with the query snippet, but adds extra functionality to the activity.

Table 7.2: Supplementary method examples

Query Code Snippet	Related Code

Query Code Snippet

```java
1  private void queueJob(final String url, final
       ImageView imageView,final Drawable
       placeholder) {
2    /* Create handler in UI thread. */
3    final Handler handler = new Handler() {
4    @Override
5    public void handleMessage(Message msg) {
6      String tag = mImageViews.get(imageView);
7        if (tag != null && tag.equals(url)) {
8          if (imageView.isShown())
9            if (msg.obj != null) {
10             imageView.setImageDrawable((Drawable)
                   msg.obj);
11           } else {
12           imageView.setImageDrawable(placeholder);
13           }
14         }
15       }
16   };
17
18   mThreadPool.submit(new Runnable() {
19     @Override
20     public void run() {
21       final Drawable bmp = downloadDrawable(url);
22       // if the view is not visible anymore, the
             image will be ready for next time in
             cache
23       if (imageView.isShown())
24       {
25         Message message = Message.obtain();
26         message.obj = bmp;
27         handler.sendMessage(message);
28       }
29     }
30   });
31 }
```

Related Code

```java
1  public void loadDrawable(final String url, final
       ImageView imageView) {
2    imageViews.put(imageView, url);
3    Drawable drawable = getDrawableFromCache(url);
4    // check in UI thread, so no concurrency issues
5    if (drawable != null) {
6      Log.d(null, "Item loaded from cache: " + url);
7      imageView.setImageDrawable(drawable);
8    } else {
9      imageView.setImageDrawable(placeholder);
10     queueJob(url, imageView);
11   }
12 }
```

Example D: Supplementary method

- The related method calls the query snippet within its method body. It is a higher-level funtionality to the query.

Query Code Snippet

```java
1  @Override
2  protected void onLayout(boolean changed, int l,
       int t, int r, int b) {
3    final int count = getChildCount();
4    for (int i = 0; i < count; i++) {
5      View child = getChildAt(i);
6      LayoutParams lp = (LayoutParams)
           child.getLayoutParams();
7      child.layout(lp.x+5, lp.y+5, lp.x +
           child.getMeasuredWidth(), lp.y +
           child.getMeasuredHeight());
8    }
9  }
```

Related Code

```java
1  @Override
2  protected LayoutParams
       generateLayoutParams(ViewGroup.LayoutParams
       p) {
3    return new LayoutParams(p);
4  }
```

Example E: Supplementary method

- The related code generates the `Layout parameters`, it will be traced by `getLayoutParam` function, which will further be called inside the query method `onLayout`. There is a dependency chain between the query and the related code.

Table 7.3: Different implementation example

Query Code Snippet	Related Code

<pre>1 public static <K, V extends Comparable<? super
      V>> SortedSet<Map.Entry<K, V>>
      entriesSortedByValues(Map<K, V> map) {
2   SortedSet<Map.Entry<K, V>> sortedEntries = new
        TreeSet<Map.Entry<K, V>>(
3   new Comparator<Map.Entry<K, V>>() {
4   @Override
5   public int compare(Map.Entry<K, V> e1,
            Map.Entry<K, V> e2) {
6   return
              e1.getValue().compareTo(e2.getValue());
7   }
8   });
9   sortedEntries.addAll(map.entrySet());
10 return sortedEntries;
11 }</pre>
	<pre>1 public static <K, V extends Comparable<? super
	      V>> Map<K, V> sortByValue(Map<K, V> map) {
	2 List<Map.Entry<K, V>> list =
	3 new LinkedList<Map.Entry<K, V>>(
	          map.entrySet());
	4 Collections.sort(list, new
	        Comparator<Map.Entry<K, V>>(){
	5 public int compare(Map.Entry<K, V> o1,
	          Map.Entry<K, V> o2){
	6 return (o1.getValue()).compareTo(
	            o2.getValue());
	7 }
	8 });
	9
	10 Map<K, V> result = new LinkedHashMap<K, V>();
	11 for (Map.Entry<K, V> entry : list){
	12 result.put(entry.getKey(), entry.getValue());
	13 }
	14 return result;
	15 }</pre>
	Example F: Different implementation
	• The question title of the SO post is: Sort the values in HashMap.
	• The query snippet from SO uses `SortedSet` to store the map entries, while the related code provides an alternative, using `LinkedList`, and show how to iterate the map.

follows:

$$Precision@k = \frac{1}{N}\sum_{i=1}^{N}\frac{|relevant_{i,k}|}{k} \tag{7.1}$$

where $|relevant_{i,k}|$ represents the number of positive related results in the top k common co-occurring results for query i, N is the number queries we evaluate, which is 50. k is the number of top results we examine, here we use $k = 1$ and $k = 3$.

We achieve 80% and 74.6% for $Precision@1$ and $Precision@3$ respectively. That is to say, for the 50 most commonly co-occurring results, 40 of them are manually examined as related, for the 150 top 3 results, 112 of them are related.

We find the following types of relevance in our sample set:

- A complementary method that adds more functionality.
- A supplementary method that helps with, or gets help from, the query.
- A different implementation for the query.

Table 7.4: Categorization of related methods

Category	Top 1	Top 3
Complementary method	20 (50%)	55 (49%)
Supplementary method	18 (45%)	53 (47%)
Different implementation	2 (5%)	4 (3%)
Total related methods	40	112

7.5.0.1 Complementary method

In this category, the query code can function alone, but the related method provides extra functionality to the query code and will further complete the user class. For the example

shown as Listing 7.1 and 7.2 in Section 7.1, the query snippet implements unzip a folder in Java. We find `zip` function. These two methods can function independently, but often implemented together to get a stronger ability for file manipulation.

Similarly, we find `decrypt` function for `encrypt` and `onPause` function for `onCreate` in Table 7.1. The two methods in each pair do not have any direct function call association between them, but they complete each other with extra functionality and are often implemented together in real-life scenarios.

Table 7.4 shows the number of related methods for each category. For the top related methods, half of them are complementary methods. 49% of the sampled top 3 related methods belong to this category.

7.5.0.2 Supplementary method

The related code serves as a helper function to the query, or vice versa. One may make function call to the other. For example the `merge` function for `sort`. `sort` calls `merge` as a helper function and cannot achieve functionality without it.

In our first example in Table 7.2, our related code `loadDrawable` calls `queueJob` inside its method body. There is another related method being recommended together with `loadDrawable`, which is shown below in Listings 7.3. `loadDrawable` also makes a function call to `getDrawableFromCache` inside its method body, The related methods give the user a broader picture of the whole class, point to a higher level of functionality the user may want to implement, and also direct the user to the most-frequently used higher level functionality and its auxiliaries.

Less than half of the sampled related results are supplementary methods.

```
1 public static Drawable getDrawableFromCache(String url) {
2 if (DrawableManager.cache.containsKey(url)) {
```

```
3 return DrawableManager.cache.get(url);
4 }
5
6 return null;
7 }
```

Listing 7.3: Related method #2

7.5.0.3 Different implementation

This category represents those related methods that have similar functionality to the query code. The result provides an alternative, or a more detailed or extended implementation for the functionality. As shown in Table 7.3, both of the methods implement sorting values in a `Map`, the query store the map entries in a `SortedSet`, while the related code uses `LinkedList`, and shows how to iterate a `Map`. For the `encrypt` function in Table 7.1, the related code also provide an alternative implementation with `String` inputs, as shown in Listing 7.4.

A small number of sampled related methods provide different implementation to the query itself.

```
1 public static String encrypt(final String password, String message) throws GeneralSecurityException {
2 try {
3 final SecretKeySpec key = generateKey(password);
4 log("message", message);
5 byte[] cipherText = encrypt(key, ivBytes, message.getBytes(CHARSET));
6 //NO_WRAP is important as was getting \n at the end
7 String encoded = String.valueOf(
8 Base64.encodeToString(cipherText, Base64.NO_PADDING ));
9 log("Base64.NO_WRAP", encoded);
10 return encoded;
11 } catch (UnsupportedEncodingException e) {
12 if (DEBUG_LOG_ENABLED)
13 Log.e(TAG, "UnsupportedEncodingException ", e);
14 throw new GeneralSecurityException(e);
15 }
16 }
```

Listing 7.4: different implementation for `encrypt`

From the in-depth manual analysis from this section, we can see that there is a large amount of related code among the common co-occurring code, and they are worth to be considered for recommendation besides similar code to the query.

7.6 Chrome Extension for Stack Overflow

In the purpose demonstrating recommending related methods in practice, we build a Chrome extension, CodeAid, for the Stack Overflow query code base and GitHub search code corpus, based on our approach of retrieving common co-occurring code. Figure 7.6 shows a screenshot of using the Chrome extension. The highlighted yellow snippet is the query method from SO, and on the right-hand side we present the ranked list of related code fragments from GitHub.

The Chrome extension demonstrates a real-life use scenario of getting related methods from GitHub for a SO snippet, thus serves as a proof of our concept of recommending related code fragments beyond similarity. The use scenario will be: suppose the user is searching on Stack Overflow for the question `how to unzip a folder in java`. The user locates the highlighted yellow snippet as the correct implementation of the query functionality. They are interested in learning the other methods that can be possibly added to their project along with the `unzip` method. So the user invokes searching on CodeAid and gets recommended related methods. They may investigate the recommended results and select to add a `zip` method to complete the functionalities of file manipulation.

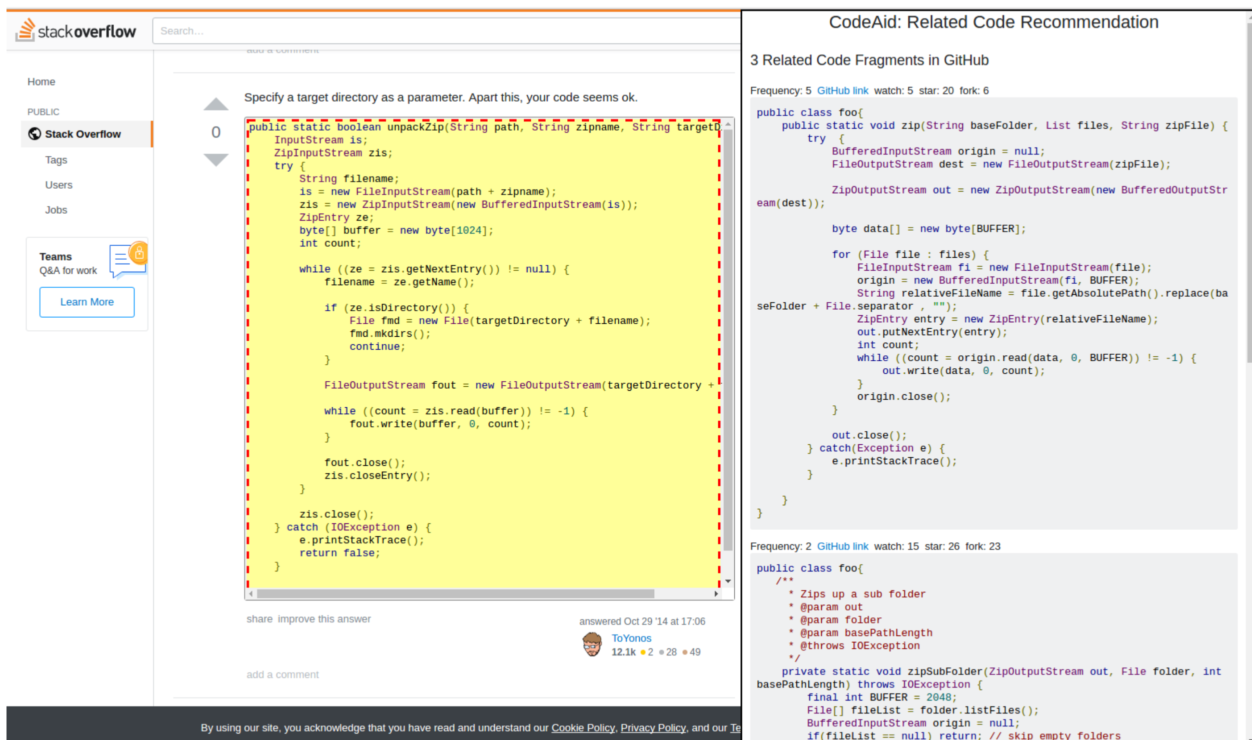


Figure 7.6: Screenshot of Chrome extension

Table 7.5: Related code which can be retrieved by FaCoY

Query Code Snippet	Related Code
1 public int readFramesChanel(short[] sampleBuffer, int offset, int numFramesToRead,int channel) throws IOException, WavFileException{	1 public int writeFrames(int[] sampleBuffer, final int offSetIn, int numFramesToWrite) throws IOException{
2 if (ioState != IOState.READING) throw new IOException("Cannot read from WavFile instance");	2 if (this.ioState != IOState.WRITING) throw new IOException("Cannot write to WavFile instance"); //$NON-NLS-1$
3 for (int f=0 ; f<numFramesToRead ; f++){	3 int offSet = offSetIn;
4 if (frameCounter == numFrames) return f;	4 for (int f = 0; f < numFramesToWrite; f++){
5 for (int c=0 ; c<numChannels ; c++){	5 if (this.frameCounter == this.numFrames) return f;
6 if(channel==c){	6 for (int c = 0; c < this.numChannels; c++){
7 sampleBuffer[offset] = (short) readSample();	7 writeSample(sampleBuffer[offSet]);
8 offset ++;	8 offSet++;
9 }	9 }
10 else	10 this.frameCounter++;
11 readSample();	11 }
12 }	12 return numFramesToWrite;
13 frameCounter ++;	13 }
14 }	
15 return numFramesToRead;	
16 }	

7.7 Comparison with code search engines

As an evaluation of CodeAid, we compare the related results of CodeAid with those from code search engines. We choose Google search engine since its the most popular destination when people look for programming assistance. We also compare to `FaCoY` [68], a code-to-code search engine which proved to have state-of-art precision. It uses SO snippets as its query base and indexes GitHub files as search space. `searchcode` [5] and `Krugle` [73] are another two online code-to-code search engines. We only compare with `FaCoY` because it beats `searchcode` and `Krugle` in the total number of outputs and the precision of outputs when using SO snippets as queries [68]. We look for whether the search engines can also retrieve top 1 related code fragments recommended by our approach in their top 10 search results.

For one out of our ten sample queries, `FaCoY` can return the related code recommended by CodeAid in its top 10 search results. This results from the related code being very similar to the query, as shown in Table 7.5. For the rest of nine queries, the related code is not

similar to the query, so it cannot be retrieved by `FaCoY`.

As for Google search engine, for five out of the ten queries, Google can locate the GitHub file(s) which contain similar methods to the query, therefore we can find the related code by our approach inside these GitHub files by manually examine the methods one by one. However, Google can only retrieve the full files, while we aim at pointing to the method which is mostly used among these files.

By performing the comparison above, we can see that code search engines may not fulfill the purpose of recommending related code as we proposed.

7.8 Threats to Validity

In terms of internal validity, we only investigate the kinds of code relevancy in common, co-occurring code, based on the observation that relevant code may often co-locate in the same file. Though we indeed find interesting types of relevant code by exploiting this proximity property of code, we may miss other kinds of relevant code that is not in the same file.

In terms of external validity, this work only analyzes code written in one programming language, Java. Previous studies have shown that JavaScript and Python have more clones than Java and C [75, 150]. Therefore, we are likely to find more common, co-occurring code for JavaScript and Python. In addition, we may find other kinds of relevant code in those different languages. For instance, Lopes et al. find that a large portion of code clones in JavaScript are generated from boilerplate [75]. Hence, we may also find that many common, co-occurring code of a JavaScript code fragment is because of boilerplate code.

Limitations. Given a code query, CodeAid needs to first search a code corpus to find the counterparts of this query and then recommends common, co-occurring code around those

counterparts. In this work, we curate a large code corpus of 50K GitHub repositories with at least five stars to ensure that we have a large code corpus to search from. However, if a given code fragment implements a project-specific logic that is not likely to be found in other projects, CodeAid is unlikely to recommend relevant code of the given code fragment. Though CodeAid currently supports Java, it is not restricted to any programming languages. The Java parser is only used to tokenize code fragments for clone detection and can be substituted with any off-the-shelf parsers of other languages.

7.9 Conclusion

Code-to-code search engines focuses on retrieving syntactically or semantically similar code fragments to a given query, without considering auxiliary or complementary code that may also be related to the query. The goal of this work is to explore the existence of other kinds of related code beyond similar code, and discuss the usefulness of such related code for recommendation in practice.

Co-occurring code is an interesting start point of exploring related code. We built an automated approach to identify, cluster, and rank common, co-occurring code. From detailed manual evaluation, we found 74% of our sampled code show valid relevancy to the query, and this relevancy can be categorized into three types–*complementary*, *supplementary*, and *alternative* functionality. Our findings shows the benefit of recommending common-occurring code, beyond similar code. We also implemented a Chrome extension that demonstrates an application of our approach in practice. We experimented to show that other search engines cannot fulfill recommending related code fragments as we proposed. We are planning to perform more user study on our tool in near future.

Chapter 8

Conclusion

There is an increasing number of research and tools that rely on software artifacts from SO and GitHub. Without understanding the artifacts themselves beforehand, those studies and tools may end up with skewed analysis results or disappointing tool performance.

This dissertation starts with investigating the quality of software artifacts from SO and GitHub. It studies the usability of SO snippets in four popular languages: Java, Python, C#, and JavaScript. The results show that usability rates for the two dynamic languages is substantially higher than that of the two statically-typed, compiled languages. With heuristic repairs, this work curates a dataset of reasonable amount of usable SO snippets for future research and development.

This dissertation then presents an exhaustive investigation of code cloning in GitHub for four of the most popular object-oriented languages: Java, C++, Python and JavaScript. The result shows the amount of file-level duplication is staggering in the four language ecosystems. Code duplication can severely skew the conclusions of the studies which use GitHub projects as data sources. The assumption of diversity of projects in those datasets may be compromised. This dissertation provides a tool to assist selecting projects from

GitHub. DéjàVu is a publicly available index of file-level code duplication. DéjàVu can help researchers and developers navigate through code cloning in GitHub, and avoid it when necessary.

SO and GitHub form a larger system of software production: the same users that reply on managing projects in GitHub often seek help on SO for implementation difficulties, and return to GitHub to apply the knowledge acquired. Because of the special bonding between the two websites, more attention is paid to the crossover between SO and GitHub in this dissertation. This dissertation conducts a detailed similarity study of code fragments bettwen the two websites, and makes publicly available a comprehensive dataset of *adaptations* and *variations* between SO and GitHub. It puts forward an adaptation taxonomy of online code examples and an automated technique for classifying adaptations. The taxonomy captures the particular kinds of adaptations done over online code examples and can be used as a guidance for future research.

The good-quality code fragments from SO and GitHub and their crossover can be leveraged in code search tools. Current code-to-code search engines focuses on retrieving syntactically or semantically similar code fragments to a given query, without considering auxiliary or complementary code that may also related to the query. The goal of the rest of this dissertation is to explore the existence of other kinds of related code beyond similar code and discuss the usefulness of such related code for recommendation in practice. Given a code query, Aroma returns extra statements which come from methods that contain similar snippets to the query, and CodeAid recommends auxiliary methods which co-occurring with similar ones to the query. Both of the two tools use selected usable SO snippets as query base. Aroma uses selected non-duplicated GitHub projects as search corpus. CodeAid starts with the crossover between SO and GitHub to investigate co-occurring methods. The evaluation of these two tools shows the benefit of recommending related code beyond similar ones, and shows that current search engines cannot fulfill recommending related code, either

statement-level or method-level, as this dissertation proposed.

Bibliography

[1] Codase. http://www.codase.com/. Accessed: 2019-05-13.

[2] Github. https://github.com/. Accessed: 2019-05-13.

[3] Google code search. https://en.wikipedia.org/wiki/Google_Code_Search. Accessed: 2019-05-13.

[4] Open hub. https://www.openhub.net/. Accessed: 2019-05-13.

[5] searchcode. https://searchcode.com/. Accessed: July 2017.

[6] *Get OS-level system information*, 2008. https://stackoverflow.com/questions/61727.

[7] *JSlider question: Position after leftclick*, 2009. https://stackoverflow.com/questions/518672.

[8] *Youtube data API : Get access to media stream and play (JAVA)*, 2011. https://stackoverflow.com/questions/4834369.

[9] *How to get IP address of the device from code?*, 2012. https://stackoverflow.com/questions/7899226.

[10] *Adding new paths for native libraries at runtime in Java*, 2013. https://stackoverflow.com/questions/15409446.

[11] *Another GitHub clone about JSlide*, 2014. https://github.com/changkon/Pluripartite/tree/master/src/se206/a03/MediaPanel.java#L343-L353.

[12] *A GitHub clone about how to get IP address from an Android device.*, 2014. https://github.com/kalpeshp0310/GoogleNews/blob/master/app/src/main/java/com/kalpesh/googlenews/utils/Utils.java#L24-L39.

[13] *A GitHub clone about JSlide*, 2014. https://github.com/changkon/Pluripartite/tree/master/src/se206/a03/MediaPanel.java#L329-L339.

[14] *A GitHub clone that adds new paths for native libraries at runtime in Java*, 2014. https://github.com/armint/firesight-java/blob/master/src/main/java/org/firepick/firesight/utils/SharedLibLoader.java#L131-L153.

[15] *A GitHub clone that downloads videos from YouTube*, 2014. https://github.com/instance01/YoutubeDownloaderScript/blob/master/IYoutubeDownloader.java#L148-L193.

[16] *A GitHub clone that gets the CPU usage*, 2014. https://github.com/jomis/nomads/blob/master/nomads-framework/src/main/java/at/ac/tuwien/dsg/utilities/PerformanceMonitor.java#L44-L63.

[17] *Stack Overflow data dump*, 2016. https://archive.org/details/stackexchange, accessed on Oct 17, 2016.

[18] L. An, O. Mlouki, F. Khomh, and G. Antoniol. Stack overflow: A code laundering platform? 03 2017.

[19] A. Arwan, S. Rochimah, and R. J. Akbar. Source code retrieval on stackoverflow using lda. In *2015 3rd International Conference on Information and Communication Technology (ICoICT)*, pages 295–299, May 2015.

[20] A. Bacchelli, L. Ponzanelli, and M. Lanza. Harnessing stack overflow for the ide. In *Proceedings of the 2012 Third International Workshop on Recommendation Systems for Software Engineering (RSSE)*, pages 562–567. ACM, 2012.

[21] K. Bajaj, K. Pattabiraman, and A. Mesbah. Mining questions asked by web developers. In *Proceedings of the 11th Working Conference on Mining Software Repositories*, MSR 2014, pages 112–121, New York, NY, USA, 2014. ACM.

[22] S. Bajracharya, T. Ngo, E. Linstead, Y. Dou, P. Rigor, P. Baldi, and C. Lopes. Sourcerer: A search engine for open source code supporting structure-based search. In *Companion to the 21st ACM SIGPLAN Symposium on Object-oriented Programming Systems, Languages, and Applications*, OOPSLA '06, pages 681–682, New York, NY, USA, 2006. ACM.

[23] S. Baltes and S. Diehl. Usage and attribution of stack overflow code snippets in github projects. *arXiv preprint arXiv:1802.02938*, 2018.

[24] B. L. Berg, H. Lune, and H. Lune. *Qualitative Research Methods for the Social Sciences*, volume 5. Pearson Boston, MA, 2004.

[25] T. F. Bissyandé, F. Thung, D. Lo, L. Jiang, and L. Réveillère. Orion: A software project search engine with integrated diverse software artifacts. In *2013 18th International Conference on Engineering of Complex Computer Systems*, pages 242–245, July 2013.

[26] S. M. Blackburn, R. Garner, C. Hoffmann, A. M. Khang, K. S. McKinley, R. Bentzur, A. Diwan, D. Feinberg, D. Frampton, S. Z. Guyer, M. Hirzel, A. Hosking, M. Jump, H. Lee, J. E. B. Moss, A. Phansalkar, D. Stefanović, T. VanDrunen, D. von Dincklage, and B. Wiedermann. The dacapo benchmarks: Java benchmarking development and

analysis. In *Proceedings of the 21st Annual ACM SIGPLAN Conference on Object-oriented Programming Systems, Languages, and Applications*, OOPSLA '06, pages 169–190, New York, NY, USA, 2006. ACM.

[27] H. Borges, A. Hora, and M. T. Valente. Understanding the factors that impact the popularity of github repositories. In *2016 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, pages 334–344, Oct 2016.

[28] J. Brandt, M. Dontcheva, M. Weskamp, and S. R. Klemmer. Example-centric programming: integrating web search into the development environment. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, pages 513–522. ACM, 2010.

[29] J. Brandt, P. J. Guo, J. Lewenstein, M. Dontcheva, and S. R. Klemmer. Two studies of opportunistic programming: interleaving web foraging, learning, and writing code. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, pages 1589–1598. ACM, 2009.

[30] M. Bruch, M. Monperrus, and M. Mezini. Learning from examples to improve code completion systems. In *Proceedings of the the 7th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on The Foundations of Software Engineering*, ESEC/FSE '09, pages 213–222, New York, NY, USA, 2009. ACM.

[31] F. J. Budinsky, M. A. Finnie, J. M. Vlissides, and P. Yu. Automatic code generation from design patterns. *IBM Systems Journal*, 35:151–171, 1996.

[32] R. P. L. Buse and W. Weimer. Synthesizing api usage examples. In *Proceedings of the 34th International Conference on Software Engineering*, ICSE '12, pages 782–792, Piscataway, NJ, USA, 2012. IEEE Press.

[33] C. Casalnuovo, P. Devanbu, A. Oliveira, V. Filkov, and B. Ray. Assert use in github projects. In *Proceedings of the 37th International Conference on Software Engineering - Volume 1*, ICSE '15, pages 755–766, Piscataway, NJ, USA, 2015. IEEE Press.

[34] W.-K. Chan, H. Cheng, and D. Lo. Searching connected api subgraph via text phrases. In *Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering*, FSE '12, pages 10:1–10:11, New York, NY, USA, 2012. ACM.

[35] S. Chatterjee, S. Juvekar, and K. Sen. Sniff: A search engine for java using free-form queries. In M. Chechik and M. Wirsing, editors, *Fundamental Approaches to Software Engineering*, pages 385–400, Berlin, Heidelberg, 2009. Springer Berlin Heidelberg.

[36] C. Chen and Z. Xing. Towards correlating search on google and asking on stack overflow. In *2016 IEEE 40th Annual Computer Software and Applications Conference (COMPSAC)*, volume 1, pages 83–92, June 2016.

[37] J. R. Cordy, T. R. Dean, and N. Synytskyy. Practical language-independent detection of near-miss clones. In *Proceedings of the 2004 Conference of the Centre for Advanced Studies on Collaborative Research*, CASCON '04, pages 1–12. IBM Press, 2004.

[38] J. R. Cordy and C. K. Roy. The nicad clone detector. In *2011 IEEE 19th International Conference on Program Comprehension*, pages 219–220, June 2011.

[39] V. Cosentino, J. L. C. Izquierdo, and J. Cabot. Findings from github: Methods, datasets and limitations. In *2016 IEEE/ACM 13th Working Conference on Mining Software Repositories (MSR)*, pages 137–141, May 2016.

[40] R. Cottrell, R. J. Walker, and J. Denzinger. Semi-automating small-scale source code reuse via structural correspondence. In *Proceedings of the 16th ACM SIGSOFT International Symposium on Foundations of software engineering*, pages 214–225. ACM, 2008.

[41] J. W. Creswell. *Research Design: Qualitative, Quantitative, and Mixed Methods Approaches*. SAGE, 2014.

[42] B. Dagenais and M. P. Robillard. Recovering traceability links between an api and its learning resources. In *2012 34th International Conference on Software Engineering (ICSE)*, pages 47–57. IEEE, 2012.

[43] D. Dig and R. Johnson. How do APIs evolve? a story of refactoring. *Journal of software maintenance and evolution: Research and Practice*, 18(2):83–107, 2006.

[44] R. Dyer, H. A. Nguyen, H. Rajan, and T. N. Nguyen. Boa: A language and infrastructure for analyzing ultra-large-scale software repositories. In *Proceedings of the 2013 International Conference on Software Engineering*, ICSE '13, pages 422–431, Piscataway, NJ, USA, 2013. IEEE Press.

[45] J.-R. Falleri, F. Morandat, X. Blanc, M. Martinez, and M. Monperrus. Fine-grained and accurate source code differencing. In *Proceedings of the 29th ACM/IEEE international conference on Automated software engineering*, pages 313–324. ACM, 2014.

[46] F. Fischer, K. Böttinger, H. Xiao, C. Stransky, Y. Acar, M. Backes, and S. Fahl. Stack overflow considered harmful? the impact of copy&paste on android application security. In *Security and Privacy (SP), 2017 IEEE Symposium on*, pages 121–136. IEEE, 2017.

[47] M. Fischer, J. Oberleitner, J. Ratzinger, and H. Gall. Mining evolution data of a product family. In *MSR '05: Proceedings of the 2005 International Workshop on Mining Software Repositories*, pages 1–5, New York, NY, USA, 2005. ACM.

[48] B. Fluri and H. C. Gall. Classifying change types for qualifying change couplings. In *ICPC '06: Proceedings of the 14th IEEE International Conference on Program Comprehension*, pages 35–45, Washington, DC, USA, 2006. IEEE Computer Society.

[49] M. Fowler. *Refactoring: Improving the Design of Existing Code*. Addison-Wesley Professional, 2000.

[50] G. Frederick, P. Bond, and S. Tilley. Vulcan: A tool for automatically generating code from design patterns. In *Proceedings of the 2nd Annual IEEE Systems Conference*, pages 1–4, 2008.

[51] M. Gabel and Z. Su. A study of the uniqueness of source code. In *Proceedings of the Eighteenth ACM SIGSOFT International Symposium on Foundations of Software Engineering*, FSE '10, pages 147–156, New York, NY, USA, 2010. ACM.

[52] R. E. Gallardo-Valencia and S. Elliott Sim. Internet-scale code search. In *Proceedings of the 2009 ICSE Workshop on Search-Driven Development-Users, Infrastructure, Tools and Evaluation*, SUITE '09, pages 49–52, Washington, DC, USA, 2009. IEEE Computer Society.

[53] J. Y. Gil and I. Maman. Micro patterns in java code. *ACM SIGPLAN Notices*, 40(10):97–116, 2005.

[54] M. Godfrey and Q. Tu. Tracking structural evolution using origin analysis. In *Proceedings of the international workshop on Principles of software evolution*, pages 117–119. ACM, 2002.

[55] M. W. Godfrey and L. Zou. Using origin analysis to detect merging and splitting of source code entities. *IEEE Transactions on Software Engineering*, 31(2):166–181, 2005.

[56] J. M. Gonzalez-Barahona, G. Robles, and S. Dueñas. Collecting data about floss development: The flossmetrics experience. In *Proceedings of the 3rd International Workshop on Emerging Trends in Free/Libre/Open Source Software Research and Development*, FLOSS '10, pages 29–34, New York, NY, USA, 2010. ACM.

[57] G. Gousios. The ghtorent dataset and tool suite. In *Proceedings of the 10th Working Conference on Mining Software Repositories*, MSR '13, pages 233–236, Piscataway, NJ, USA, 2013. IEEE Press.

[58] L. Heinemann, F. Deißenböck, M. Gleirscher, B. Hummel, and M. Irlbeck. On the extent and nature of software reuse in open source java projects. In *International Conference on Software Reuse (ICSR)*, 2011.

[59] R. Hill and J. Rideout. Automatic method completion. In *Proceedings. 19th International Conference on Automated Software Engineering, 2004.*, pages 228–235, Sept 2004.

[60] A. Hindle, E. T. Barr, Z. Su, M. Gabel, and P. Devanbu. On the naturalness of software. In *Proceedings of the 34th International Conference on Software Engineering*, ICSE '12, pages 837–847, Piscataway, NJ, USA, 2012. IEEE Press.

[61] F. Hoffa. 400,000 Github repositories, 1 billion files, 14 terabytes of code: Spaces or tabs? 2016.

[62] R. Holmes and G. C. Murphy. Using structural context to recommend source code examples. In *Proceedings. 27th International Conference on Software Engineering, 2005. ICSE 2005.*, pages 117–125, May 2005.

[63] R. Holmes and R. J. Walker. Supporting the investigation and planning of pragmatic reuse tasks. In *Proceedings of the 29th international conference on Software Engineering*, pages 447–457. IEEE Computer Society, 2007.

[64] L. Jiang, G. Misherghi, Z. Su, and S. Glondu. Deckard: Scalable and accurate tree-based detection of code clones. In *29th International Conference on Software Engineering (ICSE'07)*, pages 96–105, May 2007.

[65] E. Kalliamvakou, G. Gousios, K. Blincoe, L. Singer, D. M. German, and D. Damian. The promises and perils of mining github. In *Proceedings of the 11th Working Conference on Mining Software Repositories*, MSR 2014, pages 92–101, New York, NY, USA, 2014. ACM.

[66] T. Kamiya, S. Kusumoto, and K. Inoue. Ccfinder: A multilinguistic token-based code clone detection system for large scale source code. *IEEE Trans. Softw. Eng.*, 28(7):654–670, July 2002.

[67] H. Kim, Y. Jung, S. Kim, and K. Yi. Mecc: memory comparison-based clone detector. In *2011 33rd International Conference on Software Engineering (ICSE)*, pages 301–310, May 2011.

[68] K. Kim, D. Kim, T. F. Bissyandé, E. Choi, L. Li, J. Klein, and Y. L. Traon. Facoy: A code-to-code search engine. In *Proceedings of the 40th International Conference on Software Engineering*, ICSE '18, pages 946–957, New York, NY, USA, 2018. ACM.

[69] S. Kim, K. Pan, and E. E. J. Whitehead, Jr. Memories of bug fixes. In *Proceedings of the 14th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, SIGSOFT '06/FSE-14, pages 35–45, New York, NY, USA, 2006. ACM.

[70] S. Kim, K. Pan, and E. J. Whitehead Jr. Micro pattern evolution. In *Proceedings of the 2006 international workshop on Mining software repositories*, pages 40–46. ACM, 2006.

[71] P. S. Kochhar, T. F. Bissyandé, D. Lo, and L. Jiang. Adoption of software testing in open source projects–a preliminary study on 50,000 projects. In *2013 17th European Conference on Software Maintenance and Reengineering*, pages 353–356, March 2013.

[72] R. Koschke. Survey of research on software clones. In R. Koschke, E. Merlo, and A. Walenstein, editors, *Duplication, Redundancy, and Similarity in Software*, number 06301 in Dagstuhl Seminar Proceedings, Dagstuhl, Germany, 2007. Internationales Begegnungs- und Forschungszentrum für Informatik (IBFI), Schloss Dagstuhl, Germany.

[73] K. Krugler. *Krugle Code Search Architecture*, pages 103–120. Springer New York, New York, NY, 2013.

[74] O. A. L. Lemos, S. K. Bajracharya, J. Ossher, R. S. Morla, P. C. Masiero, P. Baldi, and C. V. Lopes. Codegenie: Using test-cases to search and reuse source code. In *Proceedings of the Twenty-second IEEE/ACM International Conference on Automated Software Engineering*, ASE '07, pages 525–526, New York, NY, USA, 2007. ACM.

[75] C. V. Lopes, P. Maj, P. Martins, V. Saini, D. Yang, J. Zitny, H. Sajnani, and J. Vitek. Déjàvu: A map of code duplicates on github. *Proc. ACM Program. Lang.*, 1(OOPSLA):84:1–84:28, Oct. 2017.

[76] S. Luan, D. Yang, C. Barnaby, K. Sen, and S. Chandra. Aroma: Code recommendation via structural code search. *Proceedings of the ACM on Programming Languages*, 3(OOPSLA):152, 2019.

[77] F. Lv, H. Zhang, J.-g. Lou, S. Wang, D. Zhang, and J. Zhao. Codehow: Effective code search based on api understanding and extended boolean model (e). In *Proceedings of the 2015 30th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, ASE '15, pages 260–270, Washington, DC, USA, 2015. IEEE Computer Society.

[78] L. Martie, T. D. LaToza, and A. v. d. Hoek. Codeexchange: Supporting reformulation of internet-scale code queries in context (t). In *2015 30th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 24–35, Nov 2015.

[79] C. McMillan, M. Grechanik, D. Poshyvanyk, C. Fu, and Q. Xie. Exemplar: A source code search engine for finding highly relevant applications. *IEEE Transactions on Software Engineering*, 38(5):1069–1087, Sept 2012.

[80] C. McMillan, M. Grechanik, D. Poshyvanyk, Q. Xie, and C. Fu. Portfolio: Finding relevant functions and their usage. In *Proceedings of the 33rd International Conference on Software Engineering*, ICSE '11, pages 111–120, New York, NY, USA, 2011. ACM.

[81] N. Meng, M. Kim, and K. S. McKinley. Systematic editing: Generating program transformations from an example. In *PLDI '11*, pages 329–342, San Jose, CA, 2011. ACM.

[82] N. Meng, M. Kim, and K. S. McKinley. Lase: locating and applying systematic edits by learning from examples. In *Proceedings of the 2013 International Conference on Software Engineering*, pages 502–511. IEEE Press, 2013.

[83] A. Mockus. Large-scale code reuse in open source software. In *Proceedings of the First International Workshop on Emerging Trends in FLOSS Research and Development*, FLOSS '07, pages 7–, Washington, DC, USA, 2007. IEEE Computer Society.

[84] A. Mockus. Amassing and indexing a large sample of version control systems: Towards the census of public source code history. In *2009 6th IEEE International Working Conference on Mining Software Repositories*, pages 11–20, May 2009.

[85] L. Moreno, G. Bavota, M. Di Penta, R. Oliveto, and A. Marcus. How can i use this method? In *Proceedings of the 37th International Conference on Software Engineering - Volume 1*, ICSE '15, pages 880–890, Piscataway, NJ, USA, 2015. IEEE Press.

[86] P. Morrison. Is programming knowledge related to age? an exploration of stack overflow. In *Proceedings of the 10th IEEE Working Conference on Mining Software Repositories (MSR), 2013*, pages 69–72. IEEE, 2013.

[87] S. Mover, S. Sankaranarayanan, R. B. Olsen, and B. E. Chang. Mining framework usage graphs from app corpora. In *2018 IEEE 25th International Conference on Software Analysis, Evolution and Reengineering (SANER)*, volume 00, pages 277–289, March 2018.

[88] M. Nagappan, T. Zimmermann, and C. Bird. Diversity in software engineering research. In *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering*, ESEC/FSE 2013, pages 466–476, New York, NY, USA, 2013. ACM.

[89] S. M. Nasehi, J. Sillito, F. Maurer, and C. Burns. What makes a good code example? - a study of programming q and a in stackoverflow. In *Proceedings of the 28th IEEE International Conference on Software Maintenance (ICSM)*, pages 102–111. ACM, 2012.

[90] A. T. Nguyen, M. Hilton, M. Codoban, H. A. Nguyen, L. Mast, E. Rademacher, T. N. Nguyen, and D. Dig. Api code recommendation using statistical learning from fine-grained changes. In *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, FSE 2016, pages 511–522, New York, NY, USA, 2016. ACM.

[91] A. T. Nguyen, T. T. Nguyen, H. A. Nguyen, A. Tamrawi, H. V. Nguyen, J. Al-Kofahi, and T. N. Nguyen. Graph-based pattern-oriented, context-sensitive source code completion. In *Proceedings of the 34th International Conference on Software Engineering*, ICSE '12, pages 69–79, Piscataway, NJ, USA, 2012. IEEE Press.

[92] T. Nguyen, N. Tran, H. Phan, T. Nguyen, L. Truong, A. T. Nguyen, H. A. Nguyen, and T. N. Nguyen. Complementing global and local contexts in representing api descriptions to improve api retrieval tasks. In *Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ESEC/FSE 2018, pages 551–562, New York, NY, USA, 2018. ACM.

[93] T. T. Nguyen, H. A. Nguyen, N. H. Pham, J. M. Al-Kofahi, and T. N. Nguyen. Graph-based mining of multiple object usage patterns. In *Proceedings of the the 7th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on The Foundations of Software Engineering*, ESEC/FSE '09, pages 383–392, New York, NY, USA, 2009. ACM.

[94] T. T. Nguyen, H. V. Pham, P. M. Vu, and T. T. Nguyen. Learning api usages from bytecode: A statistical approach. In *Proceedings of the 38th International Conference on Software Engineering*, ICSE '16, pages 416–427, New York, NY, USA, 2016. ACM.

[95] J. Noble and R. Biddle. Patterns as signs. In *Proceedings of the 16th European Conference on Object-Oriented Programming*, ECOOP '02, pages 368–391, London, UK, UK, 2002. Springer-Verlag.

[96] M. Ohtsuki, A. Makinouchi, and N. Yoshida. A source code generation support system using design pattern documents based on sgml. In *Proceedings of the Sixth Asia Pacific Software Engineering Conference*, APSEC '99, pages 292–, Washington, DC, USA, 1999. IEEE Computer Society.

[97] J. Ossher, S. Bajracharya, E. Linstead, P. Baldi, and C. Lopes. Sourcererdb: An aggregated repository of statically analyzed and cross-linked open source java projects. In *2009 6th IEEE International Working Conference on Mining Software Repositories*, pages 183–186, May 2009.

[98] J. Ossher, H. Sajnani, and C. Lopes. File cloning in open source java projects: The good, the bad, and the ugly. In *2011 27th IEEE International Conference on Software Maintenance (ICSM)*, pages 283–292, Sep. 2011.

[99] T. Parr. *The Definitive ANTLR 4 Reference*. Pragmatic Bookshelf, 2 edition, 2013.

[100] K. Philip, M. Umarji, M. Agarwala, S. E. Sim, R. Gallardo-Valencia, C. V. Lopes, and S. Ratanotayanon. Software reuse through methodical component reuse and amethodical snippet remixing. In *Proceedings of the ACM 2012 Conference on Computer Supported Cooperative Work*, CSCW '12, pages 1361–1370, New York, NY, USA, 2012. ACM.

[101] L. Ponzanelli, A. Bacchelli, and M. Lanza. Seahawk: Stack overflow in the ide. In *2013 35th International Conference on Software Engineering (ICSE)*, pages 1295–1298, May 2013.

[102] L. Ponzanelli, G. Bavota, M. D. Penta, R. Oliveto, and M. Lanza. Mining stackoverflow to turn the ide into a self-confident programming prompter. In *Proceedings of the 10th IEEE Working Conference on Mining Software Repositories (MSR), 2014*, pages 102–111. ACM, 2014.

[103] L. Ponzanelli, G. Bavota, M. D. Penta, R. Oliveto, and M. Lanza. Prompter: A self-confident recommender system. In *2014 IEEE International Conference on Software Maintenance and Evolution*, pages 577–580, Sept 2014.

[104] M. F. Porter. Readings in information retrieval. chapter An Algorithm for Suffix Stripping, pages 313–316. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1997.

[105] M. Raghothaman, Y. Wei, and Y. Hamadi. Swim: Synthesizing what i mean: Code search and idiomatic snippet synthesis. In *Proceedings of the 38th International Conference on Software Engineering*, ICSE '16, pages 357–367, New York, NY, USA, 2016. ACM.

[106] B. Ray, D. Posnett, V. Filkov, and P. Devanbu. A large scale study of programming languages and code quality in github. In *Proceedings of the 22Nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*, FSE 2014, pages 155–165, New York, NY, USA, 2014. ACM.

[107] S. P. Reiss. Semantics-based code search. In *Proceedings of the 31st International Conference on Software Engineering*, pages 243–253. IEEE Computer Society, 2009.

[108] G. Richards, A. Gal, B. Eich, and J. Vitek. Automated construction of javascript benchmarks. In *Proceedings of the 2011 ACM International Conference on Object Oriented Programming Systems Languages and Applications*, OOPSLA '11, pages 677–694, New York, NY, USA, 2011. ACM.

[109] R. Robbes and M. Lanza. How program history can improve code completion. In *2008 23rd IEEE/ACM International Conference on Automated Software Engineering*, pages 317–326, Sept 2008.

[110] C. K. Roy and J. R. Cordy. A survey on software clone detection research. *School of Computing TR 2007-541, Queen's University*, 115, 2007.

[111] C. K. Roy and J. R. Cordy. A mutation/injection-based automatic framework for evaluating code clone detection tools. In *2009 International Conference on Software Testing, Verification, and Validation Workshops*, pages 157–166, April 2009.

[112] C. K. Roy and J. R. Cordy. Near-miss function clones in open source software: an empirical study. *Journal of Software Maintenance*, 22:165–189, 2010.

[113] S. Sachdev, H. Li, S. Luan, S. Kim, K. Sen, and S. Chandra. Retrieval on source code: A neural code search. In *Proceedings of the 2Nd ACM SIGPLAN International Workshop on Machine Learning and Programming Languages*, MAPL 2018, pages 31–41, New York, NY, USA, 2018. ACM.

[114] C. Sadowski, K. T. Stolee, and S. Elbaum. How developers search for code: a case study. In *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*, pages 191–201. ACM, 2015.

[115] V. Saini, F. Farmahinifarahani, Y. Lu, P. Baldi, and C. V. Lopes. Oreo: Detection of clones in the twilight zone. In *Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ESEC/FSE 2018, pages 354–365, New York, NY, USA, 2018. ACM.

[116] H. Sajnani, V. Saini, J. Svajlenko, C. K. Roy, and C. V. Lopes. Sourcerercc: Scaling code clone detection to big-code. In *Proceedings of the 38th International Conference on Software Engineering*, ICSE '16, pages 1157–1168, New York, NY, USA, 2016. ACM.

[117] J. Saldaña. *The Coding Manual for Qualitative Researchers.*

[118] G. Salton and M. J. McGill. *Introduction to Modern Information Retrieval*. McGraw-Hill, Inc., New York, NY, USA, 1986.

[119] W. M. Shyu, E. Grosse, and W. S. Cleveland. Local regression models. In *Statistical models in S*, pages 309–376. Routledge, 2017.

[120] R. Sirres, T. F. Bissyandé, D. Kim, D. Lo, J. Klein, K. Kim, and Y. L. Traon. Augmenting and structuring user queries to support efficient free-form code search. *Empirical Software Engineering*, 23(5):2622–2654, Oct 2018.

[121] A. Sivaraman, T. Zhang, G. Van den Broeck, and M. Kim. Active inductive logic programming for code search. In *Proceedings of the 41th International Conference on Software Engineering*. IEEE Press, 2019.

[122] SPEC. Specjvm98 benchmarks, 1998.

[123] S. Subramanian and R. Holmes. Making sense of online code snippets. In *Proceedings of the 10th Working Conference on Mining Software Repositories*, MSR '13, pages 85–88, Piscataway, NJ, USA, 2013. IEEE Press.

[124] S. Subramanian, L. Inozemtseva, and R. Holmes. Live api documentation. In *Proceedings of the 36th International Conference on Software Engineering*, pages 643–652. ACM, 2014.

[125] T. Suresh, C. Luigi, A. Lerina, and D. P. Massimiliano. An empirical study on the maintenance of source code clones. *Empirical Software Engineering*, 15(1):1–34, 2010.

[126] J. Svajlenko and C. K. Roy. Evaluating clone detection tools with bigclonebench. In *Proceedings of the 2015 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, ICSME '15, pages 131–140, Washington, DC, USA, 2015. IEEE Computer Society.

[127] W. Teitelman. *PILOT: A Step towards Man-Computer Symbiosis*. PhD thesis, September 1966.

[128] F. Thung, T. F. Bissyandé, D. Lo, and L. Jiang. Network structure of social coding in github. In *Proceedings of 17th European Conference on Software Maintenance and Reengineering*, pages 323–326, March 2013.

[129] E. Torlak and S. Chandra. Effective interprocedural resource leak detection. In *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering-Volume 1*, pages 535–544. ACM, 2010.

[130] C. Treude and M. P. Robillard. Augmenting api documentation with insights from stack overflow. In *Proceedings of the 38th International Conference on Software Engineering*, ICSE '16, pages 392–403, New York, NY, USA, 2016. ACM.

[131] C. Treude and M. P. Robillard. Understanding stack overflow code fragments. In *Proceedings of the 33rd International Conference on Software Maintenance and Evolution*. IEEE, 2017.

[132] J. Tsay, L. Dabbish, and J. Herbsleb. Influence of social and technical factors for evaluating contribution in github. In *Proceedings of the 36th International Conference on Software Engineering*, ICSE 2014, pages 356–366, New York, NY, USA, 2014. ACM.

[133] Q. Tu and M. W. Godfrey. An integrated approach for studying architectural evolution. In *Program Comprehension, 2002. Proceedings. 10th International Workshop on*, pages 127–136. IEEE, 2002.

[134] Y. Ueda, T. Kamiya, S. Kusumoto, and K. Inoue. On detection of gapped code clones using gap locations. In *Ninth Asia-Pacific Software Engineering Conference, 2002.*, pages 327–336, Dec 2002.

[135] M. Umarji, S. Sim, and C. Lopes. Archetypal internet-scale source code searching. In B. Russo, E. Damiani, S. Hissam, B. Lundell, and G. Succi, editors, *Open Source Development, Communities and Quality*, volume 275 of *IFIP – The International Federation for Information Processing*, pages 257–263. Springer US, 2008.

[136] M. Umarji, S. E. Sim, and C. Lopes. Archetypal internet-scale source code searching. In *IFIP International Conference on Open Source Systems*, pages 257–263. Springer, 2008.

[137] B. Vasilescu, V. Filkov, and A. Serebrenik. Stackoverflow and github: Associations between software development and crowdsourced knowledge. In *Proceedings of 2013 International Conference on Social Computing*, pages 188–195, Sep. 2013.

[138] B. Vasilescu, D. Posnett, B. Ray, M. G. van den Brand, A. Serebrenik, P. Devanbu, and V. Filkov. Gender and tenure diversity in github teams. In *Proceedings of the 33rd Annual ACM Conference on Human Factors in Computing Systems*, CHI '15, pages 3789–3798, New York, NY, USA, 2015. ACM.

[139] B. Vasilescu, A. Serebrenik, P. Devanbu, and V. Filkov. How social q&a sites are changing knowledge sharing in open source software communities. In *Proceedings of the 17th ACM Conference on Computer Supported Cooperative Work & Social Computing*, CSCW '14, pages 342–354, New York, NY, USA, 2014. ACM.

[140] C. Vendome, G. Bavota, M. D. Penta, M. L. Vásquez, D. M. Germán, and D. Poshyvanyk. License usage and changes: a large-scale study on github. *Empirical Software Engineering*, 22:1537–1577, 2016.

[141] J. Verlaguet and A. Menghrajani. Hack: a new programming language for hhvm. `https://code.fb.com/developer-tools/hack-a-new-programming-language-for-hhvm/`, 2014.

[142] P. Wang, J. Svajlenko, Y. Wu, Y. Xu, and C. K. Roy. Ccaligner: A token based large-gap clone detector. In *Proceedings of the 40th International Conference on Software Engineering*, ICSE '18, pages 1066–1077, New York, NY, USA, 2018. ACM.

[143] S. Wang, D. Lo, and L. Jiang. An empirical study on developer interactions in stackoverflow. In *Proceedings of the 28th Annual ACM Symposium on Applied Computing*, pages 1019–1024, 2013.

[144] S. Wang, D. Lo, B. Vasilescu, and A. Serebrenik. Entagrec: An enhanced tag recommendation system for software information sites. In *Proceedings of the 2014 IEEE International Conference on Software Maintenance and Evolution*, ICSME '14, pages 291–300, Washington, DC, USA, 2014. IEEE Computer Society.

[145] M. White, M. Tufano, C. Vendome, and D. Poshyvanyk. Deep learning code fragments for code clone detection. In *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering*, ASE 2016, pages 87–98, New York, NY, USA, 2016. ACM.

[146] D. Wightman, Z. Ye, J. Brandt, and R. Vertegaal. Snipmatch: Using source code context to enhance snippet retrieval and parameterization. In *Proceedings of the 25th annual ACM symposium on User interface software and technology*, pages 219–228. ACM, 2012.

[147] E. Wong, J. Yang, and L. Tan. Autocomment: Mining question and answer sites for automatic comment generation. In *Proceedings of the 28th IEEE/ACM International Conference on Automated Software Engineering*, ASE'13, pages 562–567, Piscataway, NJ, USA, 2013. IEEE Press.

[148] Y. Wu, S. Wang, C.-P. Bezemer, and K. Inoue. How do developers utilize source code from stack overflow? *Empirical Software Engineering*, pages 1–37, 2018.

[149] D. Yang, A. Hussain, and C. V. Lopes. From query to usable code: An analysis of stack overflow code snippets. In *Proceedings of the 13th International Conference on Mining Software Repositories*, MSR '16, pages 391–402, New York, NY, USA, 2016. ACM.

[150] D. Yang, P. Martins, V. Saini, and C. Lopes. Stack overflow in github: Any snippets there? In *Proceedings of the 14th International Conference on Mining Software Repositories*, MSR '17, pages 280–290, Piscataway, NJ, USA, 2017. IEEE Press.

[151] T. Zhang, M. Song, J. Pinedo, and M. Kim. Interactive code review for systematic changes. In *Proceedings of 37th IEEE/ACM International Conference on Software Engineering. IEEE*, 2015.

[152] T. Zhang, G. Upadhyaya, A. Reinhardt, H. Rajan, and M. Kim. Are code examples on an online q&a forum reliable?: A study of api misuse on stack overflow. In *Proceedings of the 40th International Conference on Software Engineering*, ICSE '18, pages 886–896, New York, NY, USA, 2018. ACM.

[153] T. Zhang, D. Yang, M. Kim, and C. Lopes. Analyzing and supporting adaptation of online code examples. In *Proceedings of the 41th International Conference on Software Engineering*, ICSE '19, 2019.

[154] Y. Zheng. 1.x-way architecture-implementation mapping. In *Proceedings of the 33rd International Conference on Software Engineering*, ICSE '11, pages 1118–1121, New York, NY, USA, 2011. ACM.

[155] H. Zhong, T. Xie, L. Zhang, J. Pei, and H. Mei. Mapo: Mining and recommending api usage patterns. In S. Drossopoulou, editor, *ECOOP 2009 – Object-Oriented Programming*, pages 318–343, Berlin, Heidelberg, 2009. Springer Berlin Heidelberg.

[156] J. Zhou and R. J. Walker. Api deprecation: A retrospective analysis and detection method for code examples on the web. In *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, FSE 2016, pages 266–277, New York, NY, USA, 2016. ACM.

[157] L. Zou and M. W. Godfrey. Detecting merging and splitting using origin analysis. In *null*, page 146. IEEE, 2003.